C++ 자료구조 및 알고리듬분석

김책공업종합대학출판사
주체91

C++ 자료구조 및 알고리듬분석

김책공업종합대학출판사

차　　례

제3장. 목록, 탄창, 대기렬

제4장. 나무

제5장. 하쉬법

제6장. 우선권대기렬(더미)

제7장. 정렬

제8장. 분리모임ADT

제9장. 그라프알고리듬

제10장. 알고리듬설계기술

제11장. 상환분석

머 리 말

취지와 목적

자료구조와 알고리듬분석(C++) 제2판에서는 방대한 자료를 조직화하는 방법인 자료구조와 알고리듬의 실행시간을 평가하는 방법인 알고리듬분석에 대하여 서술한다. 콤퓨터의 속도가 더욱더 빨라짐에 따라 방대한 입력자료를 처리할수 있는 프로그람이 절실하게 요구된다. 그런데 프로그람의 입구자료가 커지면 프로그람의 성능이 현저하게 떨어지기때문에 이것은 프로그람의 효과성에 대하여 더 세심한 관심을 돌릴것을 요구한다. 실제로 코드를 만들기전에 알고리듬들을 분석하여 보면 개별적인 풀이들이 적당한가 하는것을 결정할수 있다. 실례로 이 책에서는 개별적인 문제들을 잘 실현하면 어떻게 방대한 자료에 대한 처리시간이 16년으로부터 1초미만으로 줄여 지는가를 보게 된다. 그러므로 실행시간을 분석하지 않고 표현하는 알고리듬이나 자료구조란 있을수 없다. 어떤 경우에는 알고리듬의 실행시간에 영향을 주는 구체적인 내용들이 조사된다.

일단 처리산법이 결정되면 또 프로그람을 작성하여야 한다. 콤퓨터가 강력해 짐에 따라 해결하여야 할 문제들이 더 커지고 더 복잡해 지며 복잡하게 얽힌 프로그람들을 개발할것이 요구된다. 이 책의 목적은 학생들에게 좋은 프로그람작성능력과 알고리듬분석방법을 다같이 가르쳐 그들이 이러한 프로그람들을 최대한 효과적으로 개발할수 있게 하는데 있다.

이 책은 현대자료구조과정(CS7)이나 알고리듬분석의 1년제연구생과정을 대상으로 한다. 학생들은 지적자와 재귀, 객체지향프로그람작성과 같은 내용들을 비롯하여 중급프로그람작성지식과 리산수학에 대한 약간한 지식은 가지고 있어야 한다.

서술방법

비록 이 책의 내용들은 대부분 언어와는 무관계하지만 프로그람을 작성하려면 어떤 특정한 언어를 리용하여야 한다. 여기서는 책제목에서 보는것처럼 C++를 선택하였다.

C++는 유력한 체계프로그람작성언어로 출현하였다. C++는 C의 문법적인 결함들을 많이 퇴치하고 같은 부류의 자료구조를 추상자료형으로 실현하기 위한 직접적인 요소(클라스와 형타)들을 보충적으로 제공해 주고 있다.

이 책을 서술할 때 가장 어려웠던 부분은 C++에 대하여 어느 정도 포함시키겠는가

를 규정하는것이였다. C++의 특성들을 너무 많이 리용하면 리해하기 힘든 부분들이 있게 되고 또 너무 적게 리용하면 클라스에 대한 C책들보다도 적게 취급되게 된다.

여기서는 객체지향방법에 근거하여 자료들을 표현하는 방식을 취하였다. 따라서 제1판과는 다르며 책에서 계승성은 없다. 같은 부류의 자료구조는 클라스형타를 리용하여 서술하였다. 또한 C++의 어려운 측면들은 일반적으로 피하고 현재 C++표준으로 되여 있는 벡토르와 문자렬클라스들을 리용하였다. 이러한 1차클라스들을 가지고 제1판에서 리용하였던 대응하는 2차클라스들을 대신함으로써 코드들을 훨씬 간단히 하였다. 현재 모든 번역기들이 다 있는것은 아니므로 부록 B에 벡토르와 문자렬클라스를 주었는데 이것들은 직결코드들로서 실제로 리용할수 있는 클라스이다. 제1장에서는 본문에서 리용하게 될 C++의 면모를 개괄한다.

C++와 JAVA의 자료구조들의 완전한 판본은 Internet상에서 얻을수 있다. 두 언어의 병립성을 더 명백히 하기 위하여 류사한 코드변환을 리용하였다. 코드들은 UNIX체계상에서는 g++(2.7.2와 2.8.1)와 SunPro 4.0을 리용하고 Windows 95체계상에서는 Visual C++ 5.0과 6.0, Borland C++ 5.0, Code warrior Pro Release 2를 리용하여 검사하였다.

개괄

제1장에는 리산수학문제들과 재귀에 대한 개괄적인 내용들이 들어 있다. 재귀를 효과적으로 리용할수 있는 유일한 방도는 재귀를 부단히 리용하는것이다. 따라서 재귀수법은 제5장을 제외한 모든 장들에서 실례와 함께 널리 쓰고 있다. 제1장에는 기본 C++에 대한 개괄적인 내용들도 들어 있다. 그리고 C++클라스설계에서 형타와 중요한 구조들에 대한 설명도 들어 있다

제2장에서는 알고리듬분석에 대하여 고찰하였다. 이 장에서는 점근분석법과 그의 기본적인 결함을 보여 주었다. 그리고 로그실행시간에 대한 깊이 있는 설명을 비롯하여 많은 실례들을 주었다. 간단한 재귀프로그람들은 그것들을 반복적인 프로그람들로 직관적으로 변환하여 분석한다. 좀 더 복잡한 분할-통치프로그람들도 소개하고 있는데 그에 대한 몇가지 분석(재귀관계를 해결하는)은 제7장에서 구체적으로 취급한다.

제3장에서는 목록과 탄창, 대기렬을 취급한다. 여기에서는 추상자료형(ADT)을 리용한 이러한 자료구조들의 코드화와 고속실현, 그것들의 리용에 중심을 두었다. 거의 대부분 완전한 프로그람이 아니지만 련습문제에 프로그람작성을 위한 충분한 사고방법을 주었다.

제4장에서는 외부탐색나무(B-나무)를 포함한 탐색나무들에 중점을 두면서 나무에 대하여 취급하였다. 실례로서는 UNIX파일체계와 수식나무들이 리용되였으며 AVL나무

들과 펼친나무들에 대해서도 소개하였다. 탐색나무실현에 대한 더 구체적인 설명은 제12장에서 주었다. 파일압축과 경기나무와 같은 나무의 보충적인 응용은 제10장에서 찾아볼수 있다. 외부매체에 대한 자료구조들은 여러 장들에서 마지막문제로 고찰하였다.

제5장은 하쉬표와 관련된 장으로서 규모는 작다. 이 장에서는 하쉬법에 대한 몇가지 분석들을 진행하고 마지막에 확장하쉬법을 취급하였다.

제6장에서는 우선권대기렬에 대한 내용을 주었다. 여기에서는 2진더미들을 취급하고 우선권대기렬에 대한 리론적으로 흥미 있는 몇가지 실현을 보충적으로 주었다. 피보나치더미는 제11장에서 설명하고 쌍더미는 제12장에서 설명하였다.

제7장에서는 정렬에 대하여 고찰한다. 이 장은 상세한 코드작성과 분석측면에서 특색이 있다. 일반적으로 쓰는 중요한 정렬알고리듬들을 모두 취급하고 비교하였다. 삽입정렬과 쉘정렬, 더미정렬, 고속정렬들과 같은 4개의 알고리듬들은 구체적으로 분석하였다. 외부정렬은 장의 마지막에서 취급하였다.

제8장에서는 분리모임알고리듬을 그 실행시간에 대한 증명과 함께 설명하였다. 이 장은 내용이 간단한데 크루스칼알고리듬에 대하여 고찰하지 않는다면 뛰여 넘을수 있다.

제9장에서는 그라프알고리듬들을 취급하였다. 그라프는 실천적으로 흔히 제기되는 문제이고 실행시간이 자료구조들에 많이 관계되기때문에 그에 대한 알고리듬들은 흥미있다. 모든 표준알고리듬들은 대부분 해당한 자료구조와 가상코드, 실행시간에 대한 분석으로 표현한다. 이러한 문제들을 적당한 관계속에서 표현하기 위하여 복잡성리론(NP-완전성문제와 비결정성문제들과 같은)을 간단히 설명하였다.

제10장에서는 알고리듬들의 설계를 문제해결의 일반적인 수법들을 검토하는 방법으로 취급하였다. 이 장에서는 실례들을 가지고 깊이 있게 확증하였다. 이 장다음부터는 실례알고리듬들을 가상코드를 리용하여 상세하게 실현함으로써 똑똑히 리해할수 있도록 하였다.

제11장에서는 유도분석들을 취급한다. 여기에서는 제4장과 제6장 그리고 이 장에서 소개되는 피보나치더미와 같은 3개의 자료구조들이 분석되고 있다.

제12장에서는 탐색나무알고리듬들과 k차나무, 쌍더미에 대하여 취급하였다. 이 장에서는 책의 다른 장들과는 달리 탐색나무들과 쌍더미에 대한 완전하고도 상세한 실현을 주고 있다. 이 장의 절들은 교수자가 다른 장들에서 설명된 내용들에 통합할수 있도록 구성되여 있다. 실례로 이 장의 우-아래흑적나무는 제4장의 AVL나무에서 설명할수 있을것이다. 부록 A에서는 표준형타서고를 고찰하고 이 책에서 설명된 내용들이 고수준자료구조들과 알고리듬들의 서고에 어떻게 적용되고 있는가에 대하여 보여 주고 있다. 부록 B에서는 vector와 string클라스들을 서술하였다.

제1장부터 제9장까지에서는 자료구조학과목의 거의 한학기과정에 충분한 내용을 준다. 시간이 허락되면 제10장을 포함시킬수 있다. 알고리듬분석에 대한 연구생과정에서는 제7장에서 제11장까지를 취급할수 있다. 제11장에서 분석되는 개선된 자료구조들은 그 앞장들에서 쉽게 참조할수 있다. 제9장의 NP-완전성문제에 대한 설명은 과정에서 리용하기에는 너무 빈약하다. NP-완전성에 대한 Garey와 Johnson의 책을 이 책의 보충으로 리용할수 있다.

련습문제

련습문제들은 매개 장의 마지막에 책에 제시된 내용순서대로 주었다. 마지막련습문제들은 그 장의 전반적인 내용을 담고 있다. 어려운 련습문제들은 별표(*)로 표시하였으며 흥미 있는 문제들은 두개의 별표(**)로 표시하였다.

거의 모든 련습문제들에 대한 해답서는 에디손 웨쓸리 롱맨출판사에서 교수자에게 직결로 제공하여 줄수 있다. 교수자들은 에디손 웨쓸리의 지역판매대리인들과 교섭하여 해답서의 리용가능성에 대한 정보를 제공받아야 한다.

참고문헌

참고문헌들은 매개 장의 마지막에 있다. 일반적으로 참고문헌들은 그 내용을 처음으로 제안한 력사적인 시기라든지 또는 책에 서술된 문제들의 확장된 내용들과 개선된 내용들을 주고 있다. 일부 참고문헌들에서는 련습문제에 대한 해답도 주고 있다.

코드리용성

이 책의 실례프로그람코드는 ftp.awl.com의 무기명 ftp를 통하여 얻을수 있다. 또한 Word Wide Web를 통하여 얻을수도 있는데 URL은 http://www.qwl.com/cseng/이다. 이 주소의 정확한 위치는 변할수 있다.

제1장. 소개

이 장에서는 이 책의 목적과 의도에 대하여 설명하고 프로그람작성에 대한 개념과 리산수학에 대하여 간단히 개괄한다. 여기에서는 다음과 같은 문제들을 고찰한다.

- 프로그람이 상당히 큰 입력자료에 대하여 어떻게 동작하는가 하는것은 적당한 크기의 입력자료에 대한 동작만큼 중요하다는 내용
- 책의 뒤부분에서 필요한 기초적인 수학지식들에 대한 개괄
- 재귀에 대한 간단한 개괄
- 책의 전반에서 쓰는 C++언어의 몇가지 중요한 특징에 대한 개괄

제1절. 책의 목적

*N*개의 수들가운데서 *k*번째로 큰 수를 결정한다고 하자. 이것을 **선택문제**라고 한다. 프로그람작성학과정을 한두번 거친 대학생들은 이 문제를 풀기 위한 프로그람을 작성하는것이 그리 어렵지 않을것이다. 이 문제에는 아주 《명백한》 몇가지 풀기방법이 있다.

이 문제를 풀기 위한 한가지 방법은 *n*개의 수들을 배렬에 읽어 들이고 거품정렬과 같은 간단한 알고리듬을 리용하여 배렬을 내리순서로 정렬한 다음 *k*번째 위치에 있는 요소를 돌려 주는것이다.

다른 한가지 더 좋은 알고리듬은 첫 *k*개의 요소를 배렬에 읽어 들이고 그것들을 내리순서로 정렬한 다음 남아 있는 요소들을 하나씩 읽어 들이되 그것이 배렬에 있는 *k*번째 요소보다 작으면 무시하고 만일 그렇지 않으면 그것을 배렬의 정확한 위치에 배치하는것이다(이때 한개 원소는 배렬밖으로 밀려 나게 된다。). 알고리듬이 완료되면 *k*번째 위치에 있는 요소를 되돌린다.

이 두 알고리듬들은 코드작성이 간단하며 널리 리용되는 수법들이다. 이때 어느 알고리듬이 더 좋으며 중요하게는 충분히 쓸수 있는것인가 하는 문제가 자연히 제기된다.

백만개 요소를 가지는 우연적인 파일에서 *k*=500,000을 리용하여 모의하면 어느 알고리듬도 합당한 시간내에 끝나지 않는다는것을 보여 준다. 비록 매개 알고리듬들이 마침내는 정확한 결과를 얻어 낸다고 하여도 여러날동안 콤퓨터를 가동시켜야 한다. 제7장에서 또 하나의 다른 방법을 설명하는데 그에 의하면 약 1s동안에 답이 얻어 진다. 따라서 우에서 제기한 알고리듬들은 실행된다고 하여도 그것들은 둘다 좋은 알고리듬이라고

볼수 없다. 왜냐하면 세번째 알고리듬에서는 적당한 시간내에 처리할수 있는 입력크기에 대하여 이 두 알고리듬들은 완전히 비실용적이기때문이다. 두번째 문제는 상용단어찾기이다. 입력은 2차원적인 문자배렬과 단어목록으로 되여 있다. 과제는 단어찾기표에서 해당한 단어들을 찾는것이다. 이 단어들은 수평,수직,대각선상에서 임의의 방향으로 놓일수 있다. 실례로 표 1-1에는 this, two, fat, that와 같은 단어들이 들어 있다. 단어 this는 1행 1렬 즉 (1, 1)에서 시작하여 (1, 4)까지이며 two는 (1, 1)에서 (1, 3)까지, fat는 (4,1)에서 (2, 3)까지, 그리고 that는 (4, 4)에서 (1, 1)까지이다.

표 1-1. 간단한 단어맞추기

	1	2	3	4
1	t	h	i	s
2	w	a	t	s
3	o	a	h	g
4	f	g	d	t

이 문제를 푸는데도 역시 적어도 두가지 간단한 알고리듬이 있다. 한가지 방법은 단어목록에 있는 매개 단어에 대하여 순서있는 3원묶음(행, 렬, 방향)을 조사하여 그 단어가 있는가를 보는것이다. 이것은 많은 다중for순환을 포함하지만 기본적으로는 간단하다. 다른 한가지 방법은 표의 끝을 벗어 나지 않는 순서화된 매개 4원묶음(행, 렬, 방향, 문자렬의 길이)에 대하여 지적된 단어가 단어표에 있는가를 검사하는것이다. 이것도 역시 많은 다중for순환을 포함하고 있는데 만일 모든 단어의 최대문자수를 알고 있다면 시간을 약간 줄일수 있다.

이 풀기방법들은 어느것이나 코드작성이 비교적 쉬우며 일반적으로 잡지들에 나오는 많은 생활적인 단어찾기문제들을 풀수 있다. 이것들은 대체적으로 16개의 행과 16개의 렬들로 이루어 져 있는데 대략 40개정도의 단어들을 포함한다. 그런데 단어맞추기판만 있고 단어목록은 사실상 영어사전이라고 볼수 있는 경우를 생각하자. 우에서 제기한 두개의 풀기방법들은 이 문제를 푸는데 상당한 시간이 요구되므로 받아 들일수 없게 된다. 그러나 이 문제를 긴 단어목록이라고 하더라도 수초내에 풀수 있는 방법은 있다.

중요한것은 많은 문제들에서 작업프로그람의 작성을 잘하지 못하고 있는것이다. 만일 프로그람을 방대한 자료모임에서 집행시킨다면 실행시간이 문제로 된다. 이 책에서는 방대한 입력들에 대한 프로그람의 실행시간평가방법과 보다 중요하게는 실제적으로 프로그람은 작성하지 않고 두 프로그람들의 실행시간을 비교하는 방법을 고찰한다. 또한 프로그람의 실행속도를 크게 개선하고 프로그람의 문제점을 찾아 내는 수법들을 설명한다. 이 수법들은 코드의 어느 부분에 노력을 최대로 집중시켜야 하겠는가를 찾아 낼수 있게 한다.

제2절. 수학지식

이 절에서는 기억해 두거나 유도할 필요가 있는 몇가지 기본적인 식들과 증명방법들을 고찰한다.

1. 지수

$$X^A X^B = X^{A+B}$$

$$\frac{X^A}{X^B} = X^{A-B}$$

$$(X^A)^B = X^{AB}$$

$$X^N + X^N = 2X^N \neq X^{2N}$$

$$2^N + 2^N = 2^{N+1}$$

2. 로그

콤퓨터과학에서 모든 알고리듬들은 특별히 지적하지 않는한 모든 로그의 밑수는 2이다.

정의: $\log_x B = A$가 필요충분조건이면 $X^A = B$이다.

이 정의로부터 다음과 같은 여러가지 변환식들을 끌어 낼수 있다.

정리 1-1.

$$\log_A B = \frac{\log_c B}{\log_c A}; \quad A, B, C > 0,\ A \neq 1$$

증명:

$X = \log_C B$, $Y = \log_C A$, $Z = \log_A B$라고 하자. 로그식의 정의에 의해 $C^X = B$, $C^Y = A$, $A^Z = B$이다. 이 세개의 같기식을 결합하면 $B = C^X = (C^Y)^Z$를 얻는다. 따라서 $X = Y \cdot Z$로 되고 이것은 $Z = X/Y$이므로 정리는 증명되였다.

정리 1-2.

$\log AB = \log A + \log B;$ $A, B>0$

증명:

$X=\log A$, $Y=\log B$, $Z=\log AB$라고 하자. 밑수를 2라고 하면 우의 식들로부터 $2^X=A$, $2^Y=B$, $2^Z=AB$로 된다. 이 세개의 식들로부터 $2^X 2^Y=AB=2^Z$를 얻는다. 따라서 $X+Y=Z$이며 이것으로서 정리는 증명되였다.

이와 류사한 방법으로 유도될수 있는 몇가지 실용적인 식들은 다음과 같다.

$$\log A/B = \log A - \log B$$
$$\log(A^B) = B\log A$$
$$\log X < X \quad \text{(모든 X>0에 대하여)}$$
$$\log 1=0,\ \log 2=1,\ \log 1024=10,\ \log 1048576=20$$

3. 합렬

기억하기 제일 쉬운 식들은

$$\sum_{i=0}^{N} 2^i = 2^{N+1} - 1$$

과 이와 류사하게

$$\sum_{i=0}^{N} A^i = \frac{A^{N+1}-1}{A-1}$$

이다.

다음식에서 $0 < A < 1$이면

$$\sum_{i=0}^{N} A^i \le \frac{1}{1-A}$$

이며 N 이 ∞로 다가가면 그 합은 $1/(1-A)$에 수렴한다. 이것들은 《등비합렬》공식들이다.

마지막식을 $\sum_{i=0}^{N} A^i (0 < A < 1)$에 대하여 다음과 같은 방법으로 유도할수 있다.

그 합을 S라고 하면

$$S = 1 + A + A^2 + A^3 + A^4 + A^5 + \cdots$$

그러면

$$AS = A + A^2 + A^3 + A^4 + A^5 + \cdots$$

이다. 이 두개의 식을 덜면(이것은 수렴하는 합렬에 대하여서만 가능하다.) 오른변의 모든 항들이 거의 다 삭제되고

$$S - AS = 1$$

이 남는데 이것은

$$S = \frac{1}{1 - A}$$

임을 의미한다.

이와 같은 수법을 리용하여 흔히 만나게 되는 합 $\sum_{i=1}^{\infty} i/2^i$ 을 계산할수 있다. 그 합을

$$S = \frac{1}{2} + \frac{2}{2^2} + \frac{3}{2^3} + \frac{4}{2^4} + \frac{5}{2^5} + \cdots$$

로 쓰고 2배하면

$$2S = 1 + \frac{2}{2} + \frac{3}{2^2} + \frac{4}{2^3} + \frac{5}{2^4} + \frac{6}{2^5} + \cdots$$

을 얻게 되는데 이 두개의 같기식을 덜면

$$S = 1 + \frac{1}{2} + \frac{1}{2^2} + \frac{1}{2^3} + \frac{1}{2^4} + \frac{1}{2^5} + \cdots$$

이다. 따라서 $S = 2$이다.

분석할 때 일반합렬의 또 한가지 형태는 등차합렬이다. 이러한 합렬은 기본식

$$\sum_{i=1}^{N} i = \frac{N(N+1)}{2} \approx \frac{N^2}{2}$$

으로 평가할수 있다.

실례로 $2+5+8+\cdots+(3k-1)$의 합을 구하기 위하여 $3(1+2+3+\cdots+ k)-(1+1+1+\cdots+1)$으로 다시 쓰면 이것은 명백히 $3k(k+1)/2-k$이다. 또 한가지 방법은 첫 항과 마지막항을 더하고(그 합은 $3k+1$) 두번째 항과 마지막으로부터 두번째 항을 더하고(그 합은 $3k+1$) 이러한 과정을 반복한다. 이런 쌍들이 $k/2$개이므로 전체 합은 $k(3k+1)/2$로 되는데 이것은 앞에서와 같은 답이다.

다음 2개의 식들은 상당히 보기 드문것들인데 최근에 나타나고 있다.

$$\sum_{i=1}^{N} i^2 = \frac{N(N+1)(2N+1)}{6} \approx \frac{N^3}{3}$$

$$\sum_{i=1}^{N} i^k \approx \frac{N^{k+1}}{|k+1|} \qquad k \neq -1$$

$k=-1$이면 두번째 식은 타당치 못하다. 이때에는 다음의 식이 필요한데 이것은 다른 수학분야들에서보다도 콤퓨터과학에서 훨씬 많이 리용된다. 수 H_N을 조화수라고 하며 그 합을 조화합이라고 한다. 다음의 근사에서 오차는 $\gamma\approx0.57721566$으로 다가가는데 이 값을 오일레르상수라고 한다.

$$H_N=\sum_{i=1}^{N}\frac{1}{i}\approx\log_e N$$

다음식들은 가장 일반적인 대수적수법들이다.

$$\sum_{i=1}^{N}f(N)=Nf(N)$$

$$\sum_{i=n_0}^{N}f(i)=\sum_{i=1}^{N}f(i)-\sum_{i=1}^{n_0-1}f(i)$$

4. Mod연산

만일 $A-B$가 N으로 나누어 지면 A는 N을 Mod로 하여 B와 합동이라고 하며 $A\equiv B(\mathrm{mod}\ N)$로 표시한다. 이것은 직관적으로 A나 B를 N으로 나눌 때 나머지가 같다는것을 의미한다. 따라서 $81\equiv61\equiv1(\mathrm{mod}\ 10)$으로 된다. 마찬가지로 $A\equiv B(\mathrm{mod}\ N)$이면 $A+C\equiv B+C(\mathrm{mod}\ N)$이고 $AD\equiv BD(\mathrm{mod}\ N)$이다.

Mod산법을 적용하는 많은 정리들이 있는데 그 일부는 수론에서 특별한 증명이 필요하다. Mod산법은 적게 쓰며 앞에서 서술한 정리들이면 충분하다.

5. 증명방법

자료구조분석에서 명제를 증명하는 가장 일반적인 두가지 방도는 귀납에 의한 증명과 부정에 의한 증명이다(때때로 교수자만이 쓰는 공리화에 의한 증명도 있다.). 정리가 거짓이라고 증명하는 가장 좋은 방도는 그의 반례를 내놓는것 이 다 .

귀납에 의한 증명

귀납에 의한 증명은 두개의 표준적인 단계로 이루어 진다. 첫번째 단계는 기본적인 경우를 증명하는것이다. 즉 몇개의 작은 값(보통 퇴화된)들에 대하여 정리가 참이라는것을 확증하는 단계로서 거의 대부분 명백하다. 그다음에는 귀납적인 가정을 한다. 이것은

일반적으로 정리가 어떤 한계 k까지의 모든 경우에 대해서 참이라고 가정한다는것을 의미한다. 그다음 이 가정을 리용하여 그다음의 값 $k+1$에 대해서도 정리가 참으로 된다는것을 보여 주면 증명이 끝난다(k가 유한한 경우에 대하여).

실례로 **피보나치수렬** $F_0=1$, $F_1=1$, $F_2=2$, $F_3=3$, $F_4=5, \cdots, F_i= F_{i-1}+F_{i-2}$가 $i\geqq1$에 대하여 $F_i<(5/3)^I$를 만족시키다는것을 증명하자(어떤 정의들은 합렬을 $F_0=0$으로 취하고 있는데 이것은 합렬을 밀기한것이다.). 이를 위하여 먼저 간단한 경우에 정리가 참이라는것을 확증한다.

$F_1=1<5/3$, $F_2=2<25/9$임을 증명하기는 쉽다. 이것이 기본적인 경우에 대한 증명이다. 이제 $i=1, 2, \cdots, k$에 대해서 정리가 참이라고 가정한다. 이것은 귀납적인 가정이다. 정리를 증명하려면 $F_{k+1}<(5/3)^{k+1}$이라는것을 보여 주어야 한다. 정의로부터

$$F_{k+1} = F_k + F_{k-1}$$

이고 갈기식의 오른쪽 변에 대하여 귀납적인 가정을 리용하여 다음 식을 얻는다.

$$\begin{aligned} F_{k+1} &< (5/3)^k+(5/3)^{k-1} \\ &< (3/5)(5/3)^{k+1}+(3/5)^2(5/3)^{k+1} \\ &< (3/5)(5/3)^{k+1}+(9/25)(5/3)^{k+1} \end{aligned}$$

이것을 간단히 하면

$$\begin{aligned} F_{k+1} &< (3/5+9/25)(5/3)^{k+1} \\ &< (24/25)(5/3)^{k+1} \\ &< (5/3)^{k+1} \end{aligned}$$

로 되여 정리는 증명되였다.

두번째 실례로서 다음의 정리를 증명하자.

정리 1-3.

만일 $N\geqq1$이면 $\sum_{i=1}^{N} i^2 = \frac{N(N+1)(2N+1)}{6}$이다.

증명:

귀납적으로 증명하자. 기본경우 $N=1$일 때 정리가 참이라는것은 쉽게 알수 있다. 귀납적인 가정으로서 정리가 $1\leqq k\leqq N$에 대하여 참이라고 하자. 이러한 가정하에서 정리가 $N+1$에 대해서도 참이라는것을 증명하자.

$$\sum_{i=1}^{N+1} i^2 = \sum_{i=1}^{N} i^2 + (N+1)^2$$

이다. 귀납적인 가정으로부터

$$\sum_{i=1}^{N+1} i^2 = \frac{N(N+1)(2N+1)}{6} + (N+1)^2$$
$$= (N+1)\left[\frac{N(2N+1)}{6} + (N+1)\right]$$
$$= (N+1)\frac{2N^2 + 7N + 6}{6}$$
$$= \frac{(N+1)(N+2)(2N+3)}{6}$$

으로 된다. 따라서

$$\sum_{i=1}^{N+1} i^2 = \frac{(N+1)[(N+1)+1][2(N+1)+1]}{6}$$

로 되여 정리는 증명되였다.

반례에 의한 증명

피보나치수렬에서 $F_k \leqq k^2$은 거짓이다. 이것을 증명하기 위한 가장 간단한 방법은 $F_{11}=144>11^2$을 계산하는것이다.

부정에 의한 증명

부정에 의한 증명은 주어 진 정리가 거짓이라고 가정하고 이러한 가정이 이미 주어 진 조건에 모순된다는것을 보여 주어 초기가정이 잘못되였다는것을 립증하는 방법으로 진행한다. 한가지 유명한 실례는 씨수의 개수는 무한하다는것을 증명하는것이다. 이것을 증명하기 위하여 주어 진 정리가 거짓이고 따라서 어떤 가장 큰 씨수 P_k가 있다고 가정한다. P_1, P_2, ⋯, P_k가 순서로 배렬된 씨수의 전체라고 하고 다음을 고찰하자.

$$N = P_1 P_2 P_3 \cdots P_k + 1$$

명백히 N은 P_k보다 크다. 그리고 가정에 의해 N은 씨수가 아니다. 그러나 N은 P_1, P_2,⋯ , P_k의 어느것으로도 정확히 나누어 지지 않는데 그것은 항상 나머지 1이 존재하기 때문이다. 이것은 모순으로 되는데 그것은 모든 수는 씨수이거나 씨수들의 적이기때문이다. 따라서 P_k가 가장 큰 씨수라는 초기가정은 거짓으로 되여 정리는 참이라는것을 의미한다.

제3절. 재귀에 대한 간단한 소개

우리가 잘 알고 있는 대부분의 수학함수들은 간단한 공식으로 서술된다. 실례로 공식

$$C = 5(F-32)/9$$

을 리용하여 화씨온도를 섭씨온도로 변환할수 있다. 주어 진 이 식을 C++함수로 서술하는것은 간단하다. 즉 함수선언과 대괄호를 제외하면 한개 행의 공식이 C++의 한개 행으로 변환된다.

수학함수들은 때때로 비표준적인 형식으로도 정의된다. 실례로 부아닌 옹근수들에 대하여 $f(0)=0$과 $f(x) = 2f(x-1)+x^2$을 만족시키는 함수 f를 정의하자. 이 정의로부터 $f(1) = 1$, $f(2) = 6$, $f(3) = 21$, $f(4) = 58$임을 알수 있다. 자기자체에 의하여 정의되는 함수를 재귀함수라고 한다. C++는 재귀적인 함수를 서술할수 있다.[1] C++가 제공하는것은 순전히 재귀적특징을 모방하려는 시도이라는것을 명심하여야 한다. 수학적으로 재귀인 모든 함수들이 C++의 재귀모의에 의하여 효과적으로(혹은 정확하게) 실현되는것은 아니다. 의도하는것은 재귀함수 f도 비재귀함수와 마찬가지로 적은 수의 지령으로 표현될수 있어야 한다는것이다. 프로그람 1-1에 재귀함수 f의 실현을 보여 주었다.

```
         int f( int x )
         {
/* 1*/       if( x==0 )
/* 2*/           return 0;
             else
/* 3*/           return 2 * f( x – 1 ) + x * x;
         }
```

프로그람 1-1. 하나의 재귀함수

1행과 2행은 기초경우로서 알고 있는것이다. 즉 함수값이 직접 알려 져 있으므로 재귀에 의거하지 않는다. $f(0)=0$이라는 사실을 반영함이 없이는 $f(x) = 2f(x-1) + x^2$와 같이 선언하는것은 수학적으로 무의미하다. 다시말하여 C++의 재귀함수는 기초경우가 없으면 의미가 없다. 3행은 재귀호출을 진행한다.

재귀에는 몇가지 중요하면서도 혼돈할수 있는 점들이 있다. 한가지 공통된 의문은 이것이 정말 순환론리인가 하는것이다. 그 대답은 비록 함수가 그자체에 의하여 정의된

---

[1] 수값계산에 재귀를 리용하는것은 일반적으로 좋지 못한것이다. 때문에 기본적인 문제점들만 서술한다.

다고 하더라도 함수의 개별적인 값들은 그자신에 의하여 정의되지 않는다는것이다. 다시 말하면 $f(5)$를 계산하여 $f(5)$를 평가한다면 순환적이다. $f(4)$가 $f(5)$의 계산결과에 의하여 평가되지 않는 한 $f(4)$에 의해 $f(5)$를 얻는것은 물론 비순환적이다. 가장 중요한 두가지 문제는 대체로《어떻게》그리고《왜》라고 하는 의문일것이다. 제3장에서《어떻게》그리고《왜》라는 문제가 명백히 해결된다. 아래에서는 불완전하게나마 설명을 준다.

재귀호출은 다른 함수호출과 마찬가지로 조종된다. 만일 f가 값 4를 가지고 호출되면 3행은 $2*f(3)+4*4$의 계산을 요구한다. 따라서 $f(3)$을 계산하게 되는데 이것은 $2*f(2)+3*3$를 계산할것을 요구한다. 그러므로 $f(2)$를 계산하기 위한 또 다른 호출이 있게 된다. 이것은 $2*f(1)+2*2$를 계산한다는것을 의미한다. 그러므로 $f(1)$이 $2*f(0)+1*1$에 의하여 계산된다. 이제는 $f(0)$을 얻어야 하는데 이것은 기초경우이므로 $f(0)=0$이라는것을 이미 알고 있다. 이것은 $f(1)$에 대한 계산의 완성을 의미하는데 현재 그 값은 1이다. 그러면 $f(2)$, $f(3)$ 최종적으로 $f(4)$를 결정할수 있다. 미결함수호출(그것들은 시작은 했지만 재귀호출이 끝날 때까지 기다려야 하는)을 그 변수들과 함께 보관해 두는 작업은 콤퓨터에 의해 자동적으로 수행된다. 그런데 중요한것은 기초경우에 도달할 때까지 재귀호출을 계속해야 한다는것이다. 실례로 $f(-1)$을 계산하려는 시도는 $f(-2)$, $f(-3)$, …을 호출하는 결과를 가져 온다. 이것은 아무리해도 기초경우에 도달하지 못하므로 프로그람은 그 답(그것은 어떤 식으로도 정의되여 있지 않다.)을 계산할수 없게 된다. 때로는 훨씬 더 미묘한 오유를 범하게 된다. 그것을 프로그람 1-2에 보여 주었다. 프로그람 1-2에서의 오유는 3행에서 bad(1)이 bad(1)이라고 정의되는것이다. 명백히 이것은 bad(1)이 실제로 무엇인가에 대한 아무러한 실마리도 주지 않고 있다. 콤퓨터는 그 값을 얻기 위하여 bad(1)을 다시 반복하여 호출한다. 결국 호출처리계는 작업공간을 벗어 나게 되며 프로그람은 비정상적으로 끝나게 된다. 일반적으로 이 함수는 하나의 특별한 경우를 제외하고는 정확히 동작한다고 말할수 있다. 그러나 여기서는 그렇지 않다. 왜냐하면 bad(2)가 bad(1)을 호출하기때문이다. 따라서 bad(2)도 역시 계산할수 없다. 더 나아가서 bad(3), bad(4), bad(5)도 모두 bad(2)를 호출하는데 bad(2)를 계산할수 없으므로 이 값들중 어느것도 얻을수 없다. 실제로 이 프로그람은 0을 제외하고 어떤 값 n에 대해서도 동작하지 않는다. 재귀프로그람들에서는《특수한 경우》란 없다.

```
          int bad( int n)
          {
/* 1*/        if(n = = 0)
/* 2*/            return 0;
              Else
/* 3*/            return bad( n / 3  +  1) + n –1;
          }
```

프로그람 1-2. 끝나지 못하는 재귀함수

이러한 고찰은 재귀에 대한 두가지 기본규칙을 보여 준다.

① 기초경우. 언제나 재귀가 없이 풀수 있는 몇가지 기초경우가 있어야 한다.
② 진행방향. 재귀적으로 풀어야 할 경우 재귀호출은 언제나 기초경우를 향하여 진행되여야 한다,

이 책은 문제를 푸는데서 재귀를 리용하게 된다. 비수학적으로 리용하는 한가지 실례로 큰 사전을 생각해 보자. 사전에 있는 단어들은 다른 단어들에 의해 정의된다. 어떤 단어를 찾아 볼 때 보통 그 정의를 그 즉시에 리해하지 못하면 그 정의에 리용된 단어들을 다시 찾아 볼수 있다. 또다시 그 일부를 리해하지 못할수도 있는데 이때에는 한동안 사전에서 탐색을 계속하여야 한다. 사전은 한계가 있으므로 첫째로, 사용자가 어떤 정의에 나오는 모든 단어들을 리해한 상태에 도달하던가(그 정의를 리해하고 지금까지의 정의의 경로로 되돌아 간다.) 둘째로, 정의들이 순환적이여서 헤여 날수 없게 되든가 정의를 리해하는데 필요한 일부 단어가 사전에 없다는것을 알게 된다.

단어들을 리해하기 위한 재귀적인 수법은 다음과 같다. 즉 사용자가 어떤 단어의 의미를 알게 되면 처리를 끝내고 그렇지 않으면 사전에서 그 단어를 찾는다. 만일 정의에 있는 모든 단어를 리해하면 처리를 끝내고 그렇지 않으면 모르는 단어들을 재귀적으로 찾아 냄으로써 그 정의가 무엇을 의미하는가 하는 표상을 가지게 된다. 이 처리는 사전이 잘 정의되여 있으면 끝나게 되지만 어떤 단어가 정의되여 있지 않거나 순환적으로 정의되여 있으면 무한순환에 빠지게 된다.

수의 출력

정의옹근수 n을 출력하려고 한다고 하자. 그 처리루틴을 printOut(n)이라고 하되 이 루틴은 다만 한자리수만 취하여 말단에 출력하는 I/O처리루틴밖에 리용할수 없다고 하자. 이 루틴을 printDigit로 호출한다. 실례로 printDigit(4)는 말단에 4를 출력한다.

재귀적인 수법은 이 문제를 아주 명백하게 처리한다. 76234를 출력하기 위하여 먼저 7623을 출력하고 그다음에 4를 출력한다. 여기서 두번째 걸음은 printDigit(n%10)로서 쉽게 완성되지만 첫번째 걸음은 본래의 문제보다 별로 더 간단해 보이지 않는다. 실제로 이것은 같은 문제이므로 printOut(n/10)을 가지고 재귀적으로 처리한다. 이것은 일반적인 재귀문제를 처리하는 방법을 주는데 여전히 프로그람이 무한순환에 빠지지 않는다는 담보가 있어야 한다. 기초경우가 정의되지 않았으므로 아직 무엇인가 더 해야 한다는것은 명백하다. 때문에 기초경우를 $0 \leq n < 10$이면 printDigit(n)이라고 해놓겠다. 그러면 printOut(n)은 0~9까지의 모든 정의 옹근수에 대하여 정의되며 그보다 큰 수들은 더 작은 정의 옹근수들로 정의된다. 이렇게 하여 순환에서 벗어 난다.[2] 총체적인 처리과정을 프로그람 1-3에 보여 주었다.

---

[2] 항목처리는 void를 되돌리는 함수에 포함된다.

```
void printOut( int n )        // Print nonnegative n
{
       if( n >= 10 )
              printOut( n / 10 );
       printDigit( n % 10 );
}
```

프로그람 1-3. 옹근수를 출력하는 재귀루틴

이것을 효과적으로 처리하기 위한 노력은 하지 않았다. 즉 Mod연산은 피했어야 하였다. 이 연산은 $n\%10 = n - \lfloor n/10 \rfloor * 10$ 이므로 대단히 비효률적이다.[3]

재귀와 귀납법

재귀적인 수자인쇄프로그람의 작업을 엄밀하게 따져 보자. 이를 위해 귀납법을 리용한다.

정리 1-4.

재귀적인 수자인쇄알고리듬은 $n \geqq 0$에 대해서 정확히 처리된다.

증명: (n의 자리수에 대한 귀납법을 리용하여)
먼저 n이 한자리수자이라면 프로그람은 명백히 정확하다. 그것은 단지 printDigit를 한번 호출하기때문이다. 이제 printOut가 k 또는 그보다 작은 자리의 모든 수자에 대하여 동작한다고 하자. k+1개 자리를 가지는 수자는 첫 k개의 수자들과 제일 작은 의미 있는 수자로 표현한다. 그러나 첫 k개의 자리들에 의하여 이루어 지는 수는 정확히 $\lfloor n/10 \rfloor$으로서 그것은 귀납적인 가정에 의해 정확히 출력되며 마지막자리의 수자는 n mod 10으로 출력된다. 따라서 프로그람은 임의의 (k+1)번째 자리에 있는 수자를 정확히 출력한다. 이와 같이 귀납법에 의하여 모든 수자들이 정확히 출력된다.

이 증명방법은 알고리듬서술과 대체로 일치하기때문에 좀 이상하게 생각될수 있다. 그것은 같은 문제에 대한 더 작은 모든 실례(그것들은 기초경우에 대한 경로상에 있다.)들에서는 정확하게 작업한다고 볼수 있는 재귀프로그람설계에서 보여 준다. 재귀프로그

[3] $\lfloor x \rfloor$는 x 보다 크지 않은 가장 큰 옹근수이다.

람에서는 오직 재귀에 의해《기묘하게》얻어 지는 작은 문제들에 대한 풀이들을 주어 진 문제에 대한 풀이로 결합하는것만이 필요하다. 이에 대한 수학적인 정리는 귀납법으로 증명할수 있다. 이것은 재귀에 대한 세번째 규칙을 보여 준다.

③ 설계규칙. 재귀호출들은 모두 처리된다고 가정한다.

이 규칙은 재귀적인 프로그람을 설계할 때 일반적으로 사용자가 처리과정의 상세한 내용을 알 필요가 없고 수많은 재귀호출 등을 추적하지 않아도 된다는것을 의미하므로 중요하다. 흔히 재귀호출들의 실제적인 처리를 순차적으로 추적해 나가는것은 몹시 어렵다. 물론 많은 경우에 이것은 재귀에 대하여 효과적으로 리용할수 있게 하는데 그것은 사용자가 복잡하고 구체적인 내부처리의 밖에서 작업할수 있도록 콤퓨터가 사용자를 지원하기때문이다.

재귀를 가진 많은 문제들에서는 계산값들을 숨긴다. 이 값들은 거의 정확한데 그것은 재귀프로그람의 알고리듬설계가 단순할뿐아니라 코드를 정확히 서술하기도 쉽기때문이다. 재귀는 결코 한개의 간단한 for순환에 대한 대용으로는 되지 않는다. 제3장 제3절에서 재귀에 대한 더 구체적인 내용을 설명한다. 재귀루틴들을 서술할 때 재귀에 대한 4개의 기본적인 **재귀규칙**을 명심하는것이 아주 중요하다.

첫째: 기초경우. 사용자는 항상 재귀없이 풀수 있는 어떤 기초경우를 가져야 한다.
둘째: 처리과정. 재귀적처리를 진행하는 경우에 재귀호출은 언제나 기초경우로 향하여 처리되여야 한다.
셋째: 설계규칙. 재귀호출들은 모두 정확히 처리된다고 가정한다.
넷째: 복리규칙. 개개의 재귀호출에 의하여 같은 실례를 푸는 중복작업은 절대로 하지 않는다.

4번째 규칙(그것의 략칭과 함께)은 다음 절에서 증명하게 되는데 피보나치수들과 같이 간단한 수학함수들을 계산하는데 재귀를 리용하는것이 일반적으로 좋지 못한 방법으로 되는 리유를 밝힌다. 사용자가 이런 규칙들을 명심한다면 재귀프로그람작성은 쉽게 실현될수 있다.

제4절. C++클라스

이 책에서는 많은 자료구조들에 대해서 취급한다. 모든 자료구조들은 자료(보통 동일한 형을 가진 항목들의 모임)를 보관하고 그 모임을 조작하는 함수들을 제공하는 객체들이다. C++(또는 다른 언어들)에서 이것은 클라스를 리용하여 실현한다. 이 절에서는 C++클라스에 대하여 고찰한다.

1. Class의 기본문법

C++에서 클라스는 그의 성원들로 이루어 져 있다. 이 성원들은 자료 또는 함수들일수 있다. 이 함수들을 **성원함수**(member function)라고 한다. 클라스의 매개 실체는 객체이다. 매개 객체는 클라스에서 지적된 자료성원들을 가진다(자료성원들이 static가 아니면 당분간 지장없이 무시할수 있다). 성원함수는 객체를 처리하는데 리용된다. 때때로 성원함수들을 방법이라고도 한다.

실례로 프로그람 1-4는 IntCell클라스를 보여 준다. IntCell클라스에서 IntCell의 매개 실체 즉 IntCell객체는 storedValue라고 하는 하나의 자료성원을 포함한다. 이 클라스에서 그밖의 모든것은 방법이다. 이 실례에는 4가지 방법이 있다. 두가지 방법은 read와 write이다. 다른 두가지 방법은 구축자라고 하는 특별한 방법들이다. 이제 몇가지 열쇠단어들을 고찰하자.

```
/**
 * A class for simulating an integer memory cell
 */
class IntCell
{
     public:
         /**
          * Construct the IntCell.
          * Initial value is 0.
          */
         IntCell( )
              { storedValue = 0; }

         /**
          * Construct the IntCell.
          * Initial value is initialValue.
          */
         IntCell( int initialValue )
              { storedValue = initialValue; }
         /**
         * Return the stored value.
         */
         int read( )
              { return storedValue; }
         /**
          * Change the stored value to x.
          */
         void write ( int x )
              { storedValue = x; }
```

```
    private:
        int storedValue;
};
```

프로그램 1-4. IntCell클라스에 대한 완전한 선언

첫번째로 두개의 표식 public와 private를 보자. 이 표식들은 클라스성원들에 대한 **호출속성**을 결정한다. 이 실례에서 storedValue자료성원은 public가 아니라 private이다. public인 성원은 임의의 클라스의 모든 방법에 의하여 호출될수 있다. private인 성원은 오직 그 클라스의 성원함수들에 의해서만 호출될수 있다. 일반적으로 자료성원들은 private로 선언되며 따라서 그 클라스의 구체적인 세부에 접근하는것을 제한하며 공동으로 리용하게 될 방법들은 public로 선언한다. 이것을 **정보은폐**(information hiding)라고 한다. private자료성원을 리용하면 객체를 리용하는 프로그람의 다른 부분에 지장을 주지 않고 객체의 내부표현을 변화시킬수 있다. 이것은 객체가 public성원함수들로부터 호출되므로 보이는 부분은 변화되지 않고 남아 있기때문이다. 클라스의 사용자는 클라스가 어떻게 실현되는가 하는 구체적인 세부는 알 필요가 없다. 많은 경우에 이런 호출을 리용하면 복잡성이 조성된다. 실례로 년, 달, 날자에 대한 항목으로 날자를 보관하는 클라스에서는 년, 달, 날자를 private로 설정하면 외부에서 이 자료성원을 2001년 2월 29일과 같이 비법적인 날자로 설정하는것을 금지시킨다. 그러나 어떤 방법들은 private로 되여 내부적으로 리용할수 있다. 클라스에서 모든 성원들은 암시적으로는 private이며 따라서 처음의 public는 선택적인것이 아니다.

두번째로 2개의 **구축자**(constructor)를 보자. 하나의 구축자는 클라스의 실체가 어떻게 구축되는가를 설명하는 방법이다. 만일 구축자를 명시적으로 정의하지 않으면 언어에 내장되여 있는 기능을 리용하여 자료성원들을 초기화하는 암시적인 구축자가 자동적으로 발생된다. IntCell클라스는 2개의 구축자를 정의한다. 처음의것은 파라메터가 정의되지 않았을 때 호출된다. 두번째 구축자는 int형파라메터가 제공될 때 호출되며 그 int형파라메터를 리용하여 storedValue성원을 초기화한다.

2. 구축자의 추가적인 문법과 접근자

비록 클라스는 서술하면 동작하지만 좋은 코드를 만들기 위한 몇개의 추가적인 문법이 있다. 프로그람 1-4에 4가지 변화를 보여 주었다(설명문들을 생략하였다.). 그 차이점은 다음과 같다.

```
            /**
            * A class for simulating an integer memory cell.
            */
            class IntCell
            {
              public:
/* 1*/          explicit IntCell( int initialValue = 0 )
/* 2*/            : storedValue( initialValue ) { }
/* 3*/          int read( ) const
/* 4*/            { return storedValue; }
/* 5*/          void write( int x )
/* 6*/            { storedValue = x: }
              private:
/* 7*/          int storedValue;
            };
```

프로그람 1-5. 수정된 IntCell 클라스

암시적인 파라메터

IntCell구축자는 암시적인 파라메터를 서술한다. 결과적으로 여전히 두개의 IntCell구축자들이 정의되여 있다. 하나는 initialValue를 접수하며 다른 하나는 파라메터가 없는 구축자인데 한개의 파라메터를 가진 구축자는 initialValue가 선택적이라는것을 암시한다. 암시적인 값 0은 파라메터가 제공되지 않을 때 0이 리용된다는것을 나타낸다. 암시적인 파라메터들은 임의의 함수에서 리용될수 있지만 구축자에서는 거의 리용되지 않는다.

초기화자표

IntCell구축자는 구축자의 본체앞에서 초기화자표를 리용한다(프로그람 1-4의 2행). 이 초기화자표는 자료성원들을 직접 초기화하기 위하여 리용된다. 프로그람 1-4에는 큰 차이점이 있는데 본체에서 대입문대신에 초기화자표를 리용하면 자료형들이 복잡한 초기화를 가지는 클라스형들인 경우에 시간을 단축한다. 일부 경우 이런 구축방법이 요구된다. 실례로 어떤 자료성원이 const(객체가 구축된 다음에는 변화시킬수 없다는 의미)이면 그때 자료성원의 값은 단지 초기화자표에 의해서만 초기화될수 있다. 또한 자료성원이 파라메터가 없는 구축자를 가지지 않는 클라스자료형이면 그때 그것은 초기화자표에서 초기화되여야 한다. 제4장의 앞에서 초기화자표의 의도적인 리용에 대하여 실례를 들어 고찰한다.

explicit(명시)구축자

IntCell구축자는 explicit형이다. 한개의 파라메터를 가진 모든 구축자들은 explicit로 설정하는데 그것은 배렬에서의 형변환을 피하기 위해서이다. 다른 한편 명시적인 형변환연산이 없이 형변환을 진행하게 하는 다소 편리한 규칙들이 있다. 보통 이것은 오유를 찾기 어렵게 하고 형기능을 심히 파괴하기때문에 쓰지 않는다. 실례로 다음의것을 고찰해 보자.

```
IntCell obj;        // obj is an IntCell
obj = 37;           // should not compile: type mismatch
```

이 코드는 한개의 IntCell객체 obj를 구축하고 다음에 대입문을 처리한다. 그러나 대입문은 대입연산자의 오른쪽 변이 다른 IntCell이 아니기때문에 동작하지 않는다. 이때 obj의 쓰기방법을 대신 리용하여야 한다. 그러나 C++는 편리한 규칙들을 가진다. 보통 하나의 파라메터구축자는 암시적인 형변환을 정의하는데 이 형변환에서 대입할수 있는 림시적인 객체가 만들어 진다. 이러한 경우에 번역기는

```
obj = 37;           //should not compile: type mismatch
```

을

```
IntCell temporary = 37;
obj = temporary;
```

와 같이 번역한다.

림시변수의 구축이 한개 파라메터구축자를 리용하여 실현될수 있다는것을 주목하여야 한다. explicit의 리용은 한개 파라메터구축자가 암시적인 림시변수를 생성하는데 리용될수 없다는것을 의미한다. 따라서 IntCell의 구축자가 explicit로 선언되면 번역기는 형오유가 있다는 통보를 내보낸다.

제7장 제8절에서 편리한 규칙들을 리용한 실례들을 보게 되는데 이 규칙과는 례외적인것이다.

explicit예약어는 새로운것이고 모든 번역기들이 그것을 지원하는것은 아니다. 그러나 전처리기는 공백을 가지고 explicit의 사건들을 모두 재배치하는데 리용할수 있으며 따라서 사용자의 코드내에 explicit를 넣지 말아야 할 리유는 없다.[4]

변하지 않는 성원함수

시험하는 성원함수는 그 객체의 상태가 변하지 않으므로 **접근자**(accessor)로 된다. 상태가 변하는 성원함수는 **변경자**(mutator)이다(왜냐하면 그것은 객체의 상태가 변하기

---

[4] 다음의 지령을 리용하시오. #define explicit

때문이다.). 실례로 일반적인 집합클라스에서 isEmpty는 접근자이며 이때 makeEmpty는 변경자이다.

C++에서는 매 성원함수를 접근자와 변경자로 표시할수 있다. 그렇게 하는것은 설계공정의 중요한 부분이며 이것을 설명으로 보지 말아야 한다. 실제로 거기에는 그 의미에 대한 중요한 결과가 있게 된다. 실례로 변경자는 변하지 않는 객체들에 대해 응용될수 없다. 모든 성원함수들은 암시적으로 변경자들이다.

성원함수를 접근자로 만들기 위해서는 파라메터형목록을 끝내는 괄호뒤에 const단어를 추가하여야 한다. const는 함수표식(signature)의 한 부분이다. const는 각이한 의미로 리용될수 있다. 함수선언은 3개의 각이한 문맥으로 표시할수 있는데 닫긴괄호뒤에 있는 const는 단지 접근자임을 나타낸다. const에 대한 다른 리용은 제1장 제5절 두번째 부분과 세번째 부분에서 서술한다.

IntCell클라스에서 read는 명백히 접근자이다. 즉 IntCell의 상태가 변하지 않는다. 따라서 그것은 3행에서 변하지 않는 성원함수로 된다. 성원함수가 접근자로 표시되였음에도 불구하고 어떤 자료성원의 값을 변화시키는 처리를 하면 번역기는 오유를 발생시킨다.[5]

3. 대면부와 실현부의 분리

프로그람 1-5의 클라스는 모두 정확한 문법적인 구축자를 표시한다. 그러나 C++에서는 클라스의 대면부를 그의 실현부와 분리하는것이 더 일반적이다. 대면부는 클라스와 그의 성원(자료와 함수들)들을 표시한다. 실현부는 함수들의 실현을 제공한다.

프로그람 1-6은 IntCell클라스의 대면부를 보여 주고 프로그람 1-7은 그 실현부를 보여 주며 프로그람 1-8은 IntCell을 리용한 루틴을 보여 준다. 몇가지 중요한 점은 다음과 같다.

```
# ifndef _IntCell_H_
# define _IntCe11_H_
/**
  * A class for simulating an integer memory cell.
  */
class IntCell
{
    public;
```

[5] 자료성원들은 const가 적용되지 말아야 한다는것을 나타내기 위해 mutable로 표시할수 있다. 이것은 모든 번역기들에서 제공되지 않는 새로운 속성이다.

```
        explicit IntCe11( int initialValue = 0 );
        int read( ) const;
        void write( int x );
    private:
        int storedValue;
};
# endif
```

프로그램 1-6. IntCell.h파일에 있는 IntCell클라스의 대면부

전처리지령

일반적으로 대면부는 .h로 끝나는 파일에 배치된다. 대면부의 내용을 요구하는 원천코드는 대면부파일을 #include로 포함하여야 한다. 이 경우 이것은 실현부파일과 main을 포함하는 2개의 파일이다. 때때로 번역된 대상과제는 다른 파일을 포함하는 파일들을 가지는데 이때 파일이 번역되는 과정에 대면부가 두번 번역되는 위험이 있게 된다. 이것은 허용할수 없는것이다. 이것을 보호하기 위하여 매개 머리부파일은 클라스대면부를 읽어 들일 때 어떤 기호를 정의하는 전처리기를 리용한다. 이에 대해서는 프로그램 1-6의 첫번째와 두번째 행에서 보여 준다.

```
# include "IntCell.h"
/**
 * Construct the IntCell with initialVa1ue.
 */
IntCell::IntCell( int initialValue ) : storedValue( initia1Value )
{
}
/**
 * Return the stored value.
 */
int IntCell::read( ) const
{
    return storedValue;
}
/**
* Store x.
*/
void IntCell::write( int x )
{
    storedValue = x;
}
```

프로그램 1-7. IntCell.cpp 파일에 있는 IntCell 클라스의 실현부

```
# include “IntCell.h”
int main( )
{
    IntCell m;          // Or, IntCell m(0); but not IntCell m( );
    m.write( 5 );
    cout << “Cell contents:” << m.read( ) << endl;
    return 0;
}
```

프로그램 1-8. TestIntCell.cpp 에서 IntCell 을 리용하는 프로그람

기호이름 _IntCell_H_는 어떤 다른 파일에 있는 이름과 같지 말아야 하는데 그것은 보통 파일이름으로부터 구축된다. 대면부파일의 첫번째 행은 그 기호가 정의되지 않는가를 검사한다. 만일 정의되지 않았으면 파일을 처리하고 정의되였으면 파일(#endif에 의해 뛰여 넘는것으로써)을 처리하지 않는다. 그것은 파일이 이미 읽어 졌다는것을 알기때문이다.

범위해결연산자

일반적으로 .cpp, .cc, .c로 끝나는 실현부파일에서 매개 성원함수는 클라스의 부분을 식별하여야 한다. 한편 함수는 대역적인 범위에(몇천억개의 오유를 나타내는) 있다고 가정되여야 한다. 그 문법은 ClassName∷member이며 여기서 ∷는 범위해결연산자이다.

함수처리의 정확성

실현된 성원함수의 실체는 클라스대면부에 표시된 선언과 정확히 일치되여야 한다. 성원함수가 접근자이든(const가 붙은) 변경자이든 실체를 다시 호출한다. 따라서 프로그람 1-6과 프로그람 1-7에 있는 read실체에 대한 const가 생략되면 오유가 발생한다. 암시적인 파라메터들은 다만 대면부에서만 설정된다. 그것들은 실현부에서는 생략된다.

객체의 형선언

C++에서 객체는 정확히 기본형과 꼭같이 선언된다. 따라서 다음의 IntCell객체에 대한 선언은 정확한것이다.

```
IntCell obj1;               // Zero parameter constructor
IntCell obj2( 12 );         // One parameter constructor
```

그러나 다음의것은 정확하지 않다.

```
IntCell obj3 = 37;          // Constructor is explicit
IntCell obj4( );            // Function declaration
```

obj3의 선언은 한개 파라메터구축자가 explicit이므로 허용될수 없다. 그러나 다른 방법으로 선언하면 옳게 된다(다시말하여 한개 파라메터구축자를 리용하는 선언은 초기값을 나타내기 위해 괄호를 리용하여야 한다.). obj4의 선언은 파라메터들을 가지지 않는 함수(다른데서 정의된)라는것을 나타내며 IntCell을 되돌린다.

4. 벡토르와 문자렬

새로운 C++표준에서는 2개의 클라스 즉 vector와 string을 정의한다. vector는 여러가지 문제점들이 있는 C++에 내장된 배렬을 치환할수 있게 한다. C++에 내장된 배렬에서 문제로 되는것은 그것이 1차클라스객체처럼 동작하지 않는다는것이다. 실례로 내장된 배렬들은 =기호를 가지고 복사될수 없고 내장된 배렬은 얼마나 많은 항목들이 거기에 보관되는지 기억할수 없으며 그의 첨수연산자는 첨수가 유효한가를 검사하지 못한다. 내장된 문자렬은 간단히 문자들의 배렬인데 크고 작은 배렬을 더하는 역할을 한다. 실례로 ==기호는 두개의 내장된 문자렬들을 정확히 비교하지 못한다.

vector와 string클라스들은 STL에서 중요한 클라스객체들로서 배렬과 문자렬을 다룬다. 하나의 vector는 그것이 얼마나 큰가를 알수 있다. 2개의 string객체들은 ==, <와 같은 연산자들을 가지고 비교할수 있다. vector와 string은 둘다 =연산자로 복사할수 있다. 가능하다면 사용자는 내장된 C++배렬과 문자렬을 리용하지 말아야 한다. 왜냐하면 이것은 항상 가능하지 않기때문인데 제1장 제5절 6에서 내장된 배렬과 문자렬을 고찰하기로 한다.

유감스럽게도 vector는 첨수범위를 검사하지 않으며 또 모든 번역기들에서 검사할수도 없다. 그러나 한계검사로서 vector클라스를 쉽게 서술할수 있는데 vector의 특징에 대한 적당한 부분모임을 부록 B에 보여 주었다.

vector와 string은 리용하기 쉽다. 프로그람 1-9에 보여 준 코드는 한조의 string을 vector<string>(vector의 형선정에 주의하시오.)에로 읽어 들이는데 그때 그것들을 반대순서로 출력한다. 벡토르가 다 차면 vector의 용량을 2배로 만들기 위하여 resize산법을 리용한다. 여기에서 size산법은 vector의 크기를 되돌리는 산법이다. vector와 string클라스가 없으면 이러한 코드는 훨씬 더 복잡해 진다.

클라스는 또한 리용하기도 쉬운데 관계연산자나 같기연산자와 같은것들을 가지고 두 문자렬의 상태를 비교한다. 따라서 Str1==Str2는 문자렬의 값이 같으면 참이다. 그것은

또한 문자렬의 길이를 되돌리는 length산법도 가진다.

```
# include <iostream.h>
# include "vector.h"          // vector (our version, in Appendix B)
# include "mystring.h"        // string (our version, in Appendix B)
int main( )
{
    vector<string> v( 5 );
    int itemsRead = 0;
    string x;

    whi1e( cin >> x )
    {
        if( itemsRead == v.size( ) )
            v.resize( v.size( ) * 2 );
        v[ itemsRead++ ] = x;
    }

    for( int i = itemsRead - 1; i >= 0; i-- )
        cout << v[ i ] << end1;
    return 0;
}
```

프로그람 1-9. 벡토르클라스의 리용. 문자렬들을 읽어 그것들을 반대순서로 출력한다.

제5절. C++의 구체적인 측면

모든 언어와 마찬가지로 C++는 자기자체의 상세한 내용과 언어적특징을 가진다. 이것들의 일부를 이 절에서 고찰한다.

1. 지적자

지적자변수는 다른 객체가 들어 가 있는 주소를 보관하는 변수이다. 그것은 많은 자료구조들에서 리용되는 기본적인 수단이다. 실례로 어떤 항목들을 기억시키기 위하여 배렬을 리용할수 있는데 이때 배렬의 중간에 어떤 항목을 삽입하자면 많은 항목들을 재배치하여야 한다. 어떤 모임을 배렬로 기억시키기보다는 매 항목을 불련속적인 기억기공간에 기억시키는것이 일반적이다. 이 기억기공간은 프로그람이 실현될 때 할당된다. 매 객체는 다음 객체를 련결하는 정보를 가지고 있다. 이 련결정보가 바로 지적자변수인데 이것은 다른 객체의 기억위치를 보관한다. 이것이 바로 제3장에서 좀 더 자세히 설명하

는 일반적인 련결목록이다.

지적자를 리용하는 연산들을 설명하기 위하여 프로그람 1-8에 있는 IntCell의 동적 할당에 대하여 다시 보자. 간단한 IntCell클라스에 대하여 이 방법을 쓸 때 C++코드를 쓰면 더 좋다는 근거가 없다는것을 강조한다. 여기서는 다만 간단한 지령으로 기억기를 동적으로 할당하는것에 대해서만 본다. 뒤에서 더 복잡한 클라스들을 보게 되는데 거기서는 이 방법들이 필요하면서도 유용하다. 그 새로운 방법을 프로그람 1-10에 보여 준다.

```
            int main( )
            {
/* 1*/          intCell *m;
/* 2*/          m = new IntCell( 0 );
/* 3*/          m->write( 5 );
/* 4*/          cout << “Cell contents:” << m->read( )<< endl;
/* 5*/          delete m;
/* 6*/          return 0;
            }
```

프로그람 1-10. IntCell에 대한 지적자들을 리용하는 프로그람
(이것을 꼭 써야 할 리유는 없다.)

선언

1행은 m의 선언을 나타낸다. *는 m이 지적자변수이라는것을 표시하는데 그것은 IntCell객체에 대한 지적자를 할당한다. m의 값은 그 객체를 가리키는 주소이다. m은 이 지적자에 의해서 초기화되지 않는다. C++에서는 m이 리용되기전에 값이 할당된다는것을 확인하기 위한 그러한 검사는 실행되지 않는다(그러나 개발자들은 이 검사를 포함하여 추가적인 검사를 진행하는 제품을 만든다.). 일반적으로 초기화되지 않는 지적자들을 리용하면 프로그람이 파괴되는데 그것은 존재하지 않는 기억기위치를 호출하기때문이다. 일반적으로 1행과 2행을 포함시키거나 m을 NULL지적자로 설정하는것은 초기값들을 제공하는데 좋은 방법들이다.

동적객체만들기

2행은 객체들을 동적으로 할당하는 방법을 보여 준다. C++에서 new는 새롭게 만들어 진 객체에 대한 지적자를 되돌린다. C++에는 령파라메터구축자를 리용하여 객체를 만드는 두가지 방법이 있다.

아래의 두개 지령은 정확한것이다.

```
m = new IntCell( );        // Ok
m = new IntCell;           // Preferred in this text
```

제1장 제4절 2에서 obj4로 설명되는 문제로 하여 일반적으로 두번째 형태를 리용한다.

페품수집과 지우기

일부 언어들에서는 어떤 객체가 더이상 참조되지 않으면 자동적으로 휴지통에 들어간다. 따라서 프로그람작성자는 그에 대하여 신경을 쓰지 않는다. 그런데 C++에서는 페품수집을 하지 않는다. new에 의해 할당된 객체가 더이상 참조되지 않으면 그 객체를 리용하지 않도록 지적자에 대한 delete연산이 적용되여야 한다. 만일 그렇게 하지 않으면 그에 리용되였던 기억기를 잃어 버리게 되는데(프로그람이 끝날 때까지) 이것을 기억기류실이라고 한다. 기억기류실은 일반적으로 C++프로그람들에서 많이 발생된다. 그러나 기억기류실은 프로그람을 조금만 주의하여 작성하면 자동적으로 제거할수 있다. 중요한 하나의 규칙은 new를 리용하지 않고 자동변수를 리용하는것이다. 초기의 프로그람에서는 IntCell이 new에 의해서 할당되지 않고 그대신 국부변수로 할당되였다. 이 경우에 IntCell에 대한 기억기는 선언된 함수가 끝날 때 자동적으로 해제된다. delete연산자는 프로그람 1-10의 5행에서 보여 준다.

지적자의 대입과 비교

C++에서 지적자변수에 대한 대입과 비교는 그것을 보관하는 기억기주소인 지적자의 값에 기초하여 진행된다. 따라서 2개의 지적자변수는 그것들이 같은 대상을 가리키면 같은 값을 가진다. 만일 그것들이 서로 다른 객체를 가리키면 지어 지적된 객체들이 서로 같아도 지적자변수값은 같지 않다. 만일 lhs와 rhs가 지적자변수(형이 일치하는)들이라면 lhs=rhs의 표현은 rhs가 가리키는 동일한 객체를 lhs도 가리킨다는것을 의미한다.[6]

지적자를 통한 객체의 성원에로의 접근

만일 지적자변수가 클라스형을 가리키면 지적된 객체의 성원을 ->연산자로 호출할수 있다. 이것은 프로그람 1-10의 3행에서 설명된다.

기타 지적자연산

C++는 때때로 지적자들에 대하여 특수한 연산을 써서 자료들을 정렬할수 있는데 그 실례로 < 연산을 들수 있다. lhs와 rhs지적자들에 대하여 lhs<rhs는 lhs에 의해서 지적된 객체가 rhs에 의해서 지적된 객체보다 더 작은 기억위치에 보관될 때 참으로 된다. 때때

[6] 이 책에서는 2진연산자에 대하여 왼쪽 방향(left-hand)과 오른쪽 방향(right-hand)을 나타내기 위하여 lhs와 rhs를 리용한다.

로 이러한 구축방법을 리용하는것이 좋다. 이와 같은 례외적인 연산실례를 제7장 제8절에서 고찰한다.

중요한 연산자의 하나로 연산자의 주소 &가 있다. 이 연산자는 객체가 놓이는 기억위치를 돌려 주며 별명을 조사하는데 편리하다. 이것은 제1장 제5절 5에서 고찰한다.

2. 파라메터넘기기

C나 Java와 같은 많은 언어들은 모든 파라메터들을 값에 의한 호출을 리용하여 넘긴다. 즉 실제적인 인수가 형식파라메터에 복사된다. 그러나 C++에서의 파라메터들에는 복사를 하면 비능률적인 대단히 복잡한 객체들이 있을수 있다. 때때로 넘겨 지는 값을 변경시킬수 있게 하는것이 좋다. 이를 위하여 C++에서는 3가지 방법으로 파라메터를 넘긴다. 그런데 C++에는 리용하려는 방법을 선택하기 위한 한가지 간단한 규칙이 있다.

여기에서는 파라메터를 넘기는 세가지 방식을 arr에서 첫 n개의 옹근수들의 평균값을 되돌리는 다음의 함수선언을 가지고 설명하는데 errorFlag는 n이 arr.size()보다 크거나 1보다 작을 때 참으로 설정된다.

```
double avg( const vector<int>& arr, int n, bool& errorFlag);
```

여기서 arr는 vector<int>형으로서 cons참조변수에 의한 호출을 리용하여 넘겨 진다. n은 int형으로서 값에 의한 호출로서 넘겨 지며 bool형인 errorFlag는 참조에 의한 호출로서 넘겨 진다. 파라메터의 넘기기방식은 일반적으로 두가지 측면에서 결정할수 있다.

① 만일 형식파라메터로 실제인수의 값을 변화시키려면 사용자는 참조에 의한 호출을 리용하여야 한다.

② 한편 실제인수의 값은 형식파라메터에 의해 변화될수 없다. 만일 파라메터의 형이 표준형이면 값에 의한 호출을 리용하며 클라스형이면 일반적으로 const참조변수에 의한 호출을 리용하여 넘겨야 한다.[7]

avg의 선언에서 errorFlag는 참조변수에 의해서 넘겨 지기때문에 errorFlag의 새로운 값을 실제인수에로 되돌린다. arr와 n은 함수 avg에 의해 변경되지 않는다. arr는 클라스형이므로 const참조변수에 의해 넘겨 지며 복사하는것은 너무 품이 든다. n은 기본형이므로 값에 의해 넘겨 지며 쉽게 복사된다.

파라메터넘기기의 선택방법을 개괄하면

- 함수에 의해서 변경되지 말아야 하는 작은 객체에 대하여서는 값에 의한 넘기기가 적당하다.

---

[7] 그러나 작은 클라스형(실례로 한가지형에 의해서만 보관되는)들은 변하지 않는 참조변수에 의한 호출대신에 값에 의한 호출을 리용하여 파라메터를 넘길수 있다.

- 함수에 의해서 변경되지 말아야 하는 큰 객체들에 대해서는 const참조변수에 의한 호출이 적당하다.
- 참조변수에 의한 호출은 함수에 의해서 변경되는 모든 객체들에서 적당하다.

3. 되돌림값넘기기

객체들은 값에 의한 되돌림, const참조변수에 의한 되돌림 그리고 때로는 참조변수에 의한 되돌림을 리용하여 되돌려 질수 있다. 많은 경우에 참조변수에 의한 되돌림은 리용하지 않는다. 드물기는 하지만 그것을 리용하는 한가지 실례를 제1장 제7절 3에서 보여준다.

값에 의한 되돌림을 리용하는것은 언제나 안전하다. 그러나 되돌려 지는 객체가 클라스형이면 복사에 드는 비용(실례로 기억기소비나 복사시간 등)을 줄이기 위해서 const 참조변수에 의한 되돌림을 리용하는것이 더 좋다.[8] 그러나 그것은 되돌림문의 표현이 함수가 되돌려 진 다음에도 수명이 확고히 담보되는 경우에만 가능하다. 이것은 C++에서 아주 복잡한 부분인데 많은 번역기들은 그것을 부정확하게 리용할 때 경고통보문을 잘 내보내지 못한다.

실례로 프로그람 1-11의 코드는 배렬에서 가장 큰 문자렬(자모순으로)을 탐색하기 위한 거의 류사한 2개의 함수를 포함한다. 이 두 함수는 const참조변수에 의하여 값을 되돌린다.

```
const string & findMax( const vector<string> & a )
{
        int maxIndex = 0;
        for( int i = 1; i < a.size( ); i++ )
            if( a[ maxIndex ] < a[ i ] )
                maxIndex = i;
        return a[ maxIndex ];
}

const string & findMaxWrong( const vector<string> & a )
{
        string maxValue = a[ 0 ];
        for( int i = 1; i < a.size( ); i++ )
            if( maxValue < a[ i ] )
                maxValue = a[ i ];
        return maxValue;
}
```

프로그람 1-11. 최대문자렬을 탐색하는 2개의 함수
(첫번째 함수만 정확하다.)

[8] 여기에서 const는 되돌려 지는 객체가 처리과정에 수정될수 없다는것을 의미한다. 이것이 파라메터목록에 있는 const와 접근자를 의미하는 const의 차이점이다.

첫번째 함수 findMax는 접수할수 있는 방법으로써 a[maxIndex]표현은 findMax의 밖에서 이미전에 존재하는 vector를 의미하며 함수가 되돌려 진 다음에도 오래동안 존재한다.

두번째 함수는 그리 좋지 못하다. maxValue는 함수가 되돌려 진 다음에는 존재하지 않는 국부변수이다. 따라서 복사처리가 없이 되돌리는것은 적합하지 않다. 만일 오유통보문내보내기에서 실패하면 유용한 정보를 포함하거나 포함하지 않을수 있는데 그것은 번역기가 maxValue에 의해서 리용된 기억기수정을 어떻게 고속으로 결정하는가에 관계된다. 이것은 오유수정작업을 어렵게 한다.

4. 참조변수

참조변수나 const 참조변수는 파라메터넘기기에 많이 리용된다. 그런데 그것들은 국부변수들로서도 리용될수 있고 클라스자료성원들로서도 리용될수 있다. 이런 경우 변수이름은 그것들이 참조하는 객체들에 대하여 동의어로 된다(형식파라메터들이 참조호출에서 실제 인수들에 대하여 동의어로 되는것은 대단히 많다.). 국부변수들은 복사비용을 줄이며 따라서 클라스형집합을 포함하는 자료구조를 검색하는데 리용할수 있다. 따라서 많은 경우에

```
string x = findMax( a );
…
cout << x << endl;
```

과 같은 코드는

```
const string & x = findMax( a );
…
cout << x << endl;
```

과 같이 쓰는것이 더 효률적이다.

제5장에서 보게 되는 두번째 리용은 복잡하게 표현된 객체의 이름을 다시 다는데 국부참조변수를 쓰는것이다. 고찰하게 될 코드는 다음과 같다.

```
List<T>& whichList = theLists[ hash( x, theLists.Size( ))];
ListItr<T> itr = whichList.find( x );
if( itr.isPastEnd( ))
      whichList.insert( x, whichList.zeroth( ));
```

참조변수들을 리용하면 상당히 복잡한 표현인 theLists[hash(x, theLists.Size( ))]가 세번 서술(그리고 그때 처리되는)되는것을 대신할수 있다.

참조변수들은 클라스자료성원으로 리용될수 있는데 이 책에서는 취급하지 않는다(그러나 제3장에 있는 련습문제 3-4에서는 참조변수를 자료성원으로 리용하는 설계과제

를 제기한다). 참조변수는 참조될 때 객체에 대한 구축자에 의해 초기화되여야 한다.

5. 3대요소: 해체자, 복사구축자, 대입연산자

C++에서 클라스들은 이미전에 고찰한 3개의 특별한 함수를 가진다. 이것들이 바로 해체자, 복사구축자, 대입연산자이다. 많은 경우에 사용자는 번역기에 의해 제공되는 암시적인 처리를 리용한다.

해체자

해체자는 항상 객체를 마감짓는 기능을 수행하며 delete연산으로 서술된다. 일반적으로 해체자의 기능은 다만 객체가 리용되는동안 할당되였던 자원을 해방시켜 주는것이다. 이것은 모든 new연산자들에 대응하여 delete를 호출하는데 마치 열려 졌던 파일들을 닫는것과 같은것으로 생각할수 있다. 매 자료성원에 대하여서는 해체자를 암시적으로 적용한다.

복사구축자

같은 형의 객체를 복사하여 초기화된 새로운 객체를 구축하는 특별한 구축자가 있다. 이것을 복사구축자라고 한다. IntCell객체와 같은 어떤 객체에서 복사구축자는 다음과 같이 호출된다.

- 초기화를 가진 선언은 다음과 같다.

```
IntCell B = C;
IntCell B( C );
그러나
B = C;    // Assignment operator, discussed later
```

  는 아니다.
- 객체는 앞에서 언급한 값에 의한 호출(&나 const& 아니라)을 리용하여 넘기는데 때로는 임의의 방법으로도 넘길수 있다.
- 객체는 값(&나 const& 대신에)에 의해 되돌려 진다.

첫번째 경우는 구축되는 객체를 명백하게 요구하므로 제일 리해하기가 쉽다. 두번째와 세번째 경우는 사용자에게는 보이지 않는 림시객체를 구축한다. 그렇지만 어디까지나 구축방법이므로 이 두가지 방법으로 객체를 복사하여 새롭게 만들수 있다.

암시적으로 복사구축자는 실행시 매개 자료성원들에 복사구축자들을 적용하여 실현한다. 기본형(실례로 int, double 또는 지적자들)들을 가지는 자료성원들에 대하여서는 간

단히 대입으로 처리한다. IntCell클라스의 storedValue자료성원의 경우가 이것을 보여 준다. 그자체가 클라스객체인 자료성원들에 대해서는 매 자료성원들의 클라스에 대한 복사구축자가 그 자료성원들에 적용된다.

대입연산자 operator=

복사대입연산자 operator=는 두 객체들이 이미 구축된 다음에 적용될 때 호출된다. lhs=rhs는 rhs의 상태가 lhs에 의무적으로 복사된다. 암시적으로 대입연산자는 실행시 매개 자료성원에 operator=를 적용하여 실현한다.

암시적인 기능

IntCell클라스를 시험할 때 암시적인 값들은 완전히 접수할수 있는것이므로 사용자는 아무처리도 하지 않아도 된다. 이것은 흔히 있는 경우이다. 만일 어떤 클라스가 기본형들로만 된 자료성원들로 구성되거나 론리적으로 암시적인 값을 가지는 객체들로 구성되여 있으면 클라스의 암시적인 값은 보통 리치에 맞는다. 따라서 자료성원들이 int, double, vector<int>, string 지어 vector<string>인 클라스는 암시적인 값을 접수한다.

중요한 문제는 지적자를 가지고 있는 자료성원을 포함하는 클라스에서 제기된다. 이 문제에 대한 해결방법을 제3장에서 상세히 보기로 한다. 여기서는 이에 대하여 대략적으로만 고찰한다. 어떤 클라스가 지적자로 된 한개의 자료성원을 포함한다고 하자. 이 지적자는 동적으로 할당된 객체를 가리킨다. 지적자들에 대한 암시적인 해체자는 아무런 처리도 하지 않는다(이것이 delete를 다시 호출하여야 하는 리유이다.). 더우기 복사구축자와 operator=는 지적된 객체들을 복사하지 않고 지적자의 값만을 간단히 복사한다. 따라서 단순히 같은 객체를 가리키는 지적자들을 포함하는 2개의 클라스를 가진다. 이것을 피상적인 복사라고 한다. 일반적으로 사용자는 구체적인 복사를 기대하는데 여기에서 전체 객체의 확장이 이루어 진다. 따라서 클라스가 지적자를 자료성원들로서 가지고 있을 때 그리고 그 의미가 중요할 때 일반적으로 해체자와 대입연산자, 복사구축자들을 실현해 주어야 한다.

IntCell에서 이 연산들은

```
~IntCell( );                          //destructor
IntCell( const IntCell & rhs);        //copy constructor
const IntCell& operator = ( const IntCell& rhs );
```

와 같이 서술할수 있다.

IntCell에 대한 암시적인 값을 접수할수 있다는 사실은 프로그람 1-12에서처럼 임의의 방법으로 그것을 실현할수 있다는것을 보여 준다. 해체자는 그것이 실행된 다음에 자

료성원들에 의하여 자동적으로 호출된다. 그래서 암시적으로는 빈 본체이다. 복사구축자에서 암시적인 값들은 복사구축자들의 초기화자표에 있으며 그 뒤에 본체의 실행부가 따른다. 만일 아무런 값도 초기화자표에 없으면 복사를 진행하는것이 아니라 매개 자료성원이 암시적(령파라메터)으로 초기화된다는것에 주의하여야 한다.

operator=는 제일 흥미 있는 부분이다. 1행은 별명검사인데 그것은 자기자체를 복사하는것이 아니다. 조건이 성립되면 operator=를 매개 자료성원에 적용한다(2행에서). 이때 3행에서 현재 객체에로의 참조값을 되돌리는데 대입문들은 a=b=c로 바꿀수 있다.

우에서 서술한 루틴들에서 만일 암시적인 값들이 리용되면 그것은 언제나 접수할수 있다. 그러나 만일 암시적인 값들이 적용되지 않으면 해체자와 operator=, 복사구축자를 실현하여야 한다. 암시적인 처리가 없으면 복사구축자는 표준적인 구축을 모방하여 실현될수 있는데 이때 operator=를 호출한다. 흔히 리용되는 또 하나의 선택은 복사구축자를 적당히 실현하는것인데 그때 값에 의한 호출을 금지하기 위하여 그것을 private부분에 배치한다.

```
    IntCell::~IntCell( )
    {
        // Does nothing, since IntCell contains only an int data
        // member. If IntCell contained any class objects, their
        // destructors would be called.
     }
      IntCell::IntCe11( const IntCett & rhs ): storedVa1ue( rhs.storedValue )
      {
      }

      const IntCell & IntCell::operator=( const IntCell & rhs )
      {
/*1*/       if( this != &rhs )      // Standard alias test
/*2*/               storedValue = rhs.storedValue;
/*3*/       return *this;
       }
```

프로그람 1-12. 3대요소에 대한 암시적인 처리들

암시적인 기능을 리용하지 않는 경우

암시적인 값들을 리용하지 않는 가장 일반적인 경우는 자료성원이 지적자형인 때와 지적자변수가 어떤 객체의 성원함수에 의해 할당될 때이다(구축자와 같은). 하나의 실례

로서 프로그람 1-13에서 보여 준것처럼 int형변수를 동적으로 할당하여 IntCell을 실현한다고 하자. 간단히 고찰하기 위해 대면부와 실현부를 분리하지 않는다.

```
class IntCell
{
   public:
      explicit IntCell( int initialValue = 0 )
      { storedValue = new int( initia1Va1ue ); }
      int read( ) const
      { return *storedVa1ue; }
      void write ( int x )
      { *storedValue = x; }
   private:
      int *storedValue;
};
```

프로그람 1-13. 자료성원은 지적자변수이며 여기에서 암시적인 처리는 좋지 않다.

프로그람 1-14에는 현재 많은 문제점들이 있다. 첫째로, 출력은 4가 비록 론리적으로는 하나만 리용되였다 할지라도 3개의 4가 표시된다. 문제는 암시적으로 operator=와 복사구축자가 지적자변수 storedValue를 복사하는데 있다. 따라서 a.storedValue, b.storedValue, c.storedValue는 모두 같은 int형값을 가리킨다. 이 복사들은 피상적이다. 즉 지적자변수가 복사된다. 둘째로, 좀 명백하지 않는 문제는 기억기류실이다. 초기에 객체 a의 구축자에 의해서 할당된 int형변수는 할당된 기억기를 유지하거나 수정하여야 한다. c의 구축자에 의해 할당된 int형변수는 어떤 지적자변수에 의해 더는 참조되지 않는다. 그것은 또한 그에 대한 지적자가 더이상 필요 없으므로 갱신되여야 한다.

```
int f( )
{
      IntCell a( 2 );
      IntCell b = a;
      IntCell c;
      c = b;
      a.write( 4 );
      cout << a.read( ) << endl << b.read( ) << endl << c.read( ) << endl;
      return 0;
}
```

프로그람 1-14. 프로그람 1-13의 문제점들을 나타내는 간단한 함수

이 문제점들을 조정하기 위하여 3개의 주요기능을 실현한다. 그 결과(분리된 대면부와 실현부를 가진)를 프로그람 1-15에 보여 주었다. 일반적으로 말하면 해체자는 기억기를 갱신하기 위하여 필요한데 이때 복사대입연산자와 복사구축자에 대한 암시적인 값들은 접수될수 없다. 만일 클라스가 자기자체를 복사할수 없는 자료성원을 포함하면 암시적인 대입연산자 operator=는 동작하지 않는다. 뒤에서 이러한 몇개의 실례를 고찰하기로 한다.

```
class IntCell
{
    public:
        explicit IntCe11( int initialValue = 0 );
        IntCell( const IntCell& rhs );
        ~IntCell( );
        const IntCell& operator=( const IntCell& rhs );
        int read( ) const;
        void write ( int x );
    private:
        int *storedValue;
};
IntCell::IntCell( int initialValue )
{
        storedValue = new int( initialValue );
}
IntCe1l::IntCe1l( const IntCell & rhs )
{
        storedValue = new int( *rhs.storedValue );
}
IntCe11::~IntCell( )
{
        delete storedValue;
}
const IntCell& IntCell::operator=( const IntCell& rhs )
{
        if( this != &rhs )
        *storedVa1ue = *rhs.storedValue;
        return *this;
}
int IntCell::read( ) const
{
        return *storedVa1ue;
}
void IntCell::write( int x )
{
        *storedVa1ue = x;
}
```

프로그람 1-15. 지적자변수인 자료성원과 3대요소

6. C언어와의 대비

C++는 C로부터 기본적인 문법들을 계승한다. 일부 C문법들이 때때로 C++에서도 리용되는데 C++는 둘중의 하나를 지원한다. 그 몇가지를 보기로 하자.

구조체

C++에서 struct는 class가 암시적인 기능들을 가지고 있다는것을 제외하고는 class와 거의 류사한데 그 차이는 모든 성원들이 public이라는것이다. 다른 의미적인 차이는 없다. 결국 struct를 리용하지 않고 C++프로그람을 서술하는것이 쉽다. 여기서 struct는 일반적으로 public자료와 구축자들만을 포함하는 class를 나타내기 위하여 리용되는데 이러한 리유로 하여 class는 C로 표현된 struct와 류사하게 동작한다.

형정의

typedef는 어떤 기호 또는 기호렬이 이미 존재하고 있는 어떤 변수의 형과 갈다는것을 나타내기 위하여 리용된다. 실례로

```
typedef string* ptr_to_string;
```

은 ptr_to_stirng이 stirng*형과 동의어라는것을 지시한다. typedef는 C에서보다 C++에서 좀 적게 리용되는데 그것은 많은 경우에 typedef를 리용하는것보다 이 형의 작용을 숨기는 새로운 클라스를 정의하는것이 더 유리하기때문이다.

typedef를 리용하는데는 두가지 일반적인 경우가 있다. 하나는 체계에 관계되는 정보를 정의하는것이다. 따라서 32bit옹근수에 대응하는 형인 int32는 머리부파일에서 정의된 typedef여야 한다. 그것은 일부 체계들에서 int이여야 하고 다른데서는 short이여야 하며 또 다른데서는 long이여야 한다. 다른 하나는 long형변수의 이름에 대한 동의어를 제공하는것이다. long형이름들은 형판들이 례를 들어 설명될 때 일반적으로 많이 쓰이는데 (STL에서 특별히) 이러한 실례들을 부록 A에 주었다.

C형태의 파라메터넘기기

C에서 모든 파라메터들이 값에 의한 호출을 리용하여 넘겨 진다. C프로그람작성자들은 흔히 참조에 의한 호출을 리용하여 파라메터를 넘길것을 요구한다. C에서는 이것이 불가능하므로 일반적으로 객체대신에 객체에 대한 지적자를 넘기는 묘한 수법을 리용한다. 값에 의한 호출은 지적자(그 값을 가리키는)의 값이 변할수 없다는것을 의미하지만 지적자변경을 금지하지는 않는다. 이 표현형식을 서술하기 위하여 옹근수를 참조에

의해서 어떻게 넘기는가 하는 방법을 보자. zero함수는 지적된 객체를 0으로 변경시킨다. zero의 선언은 다음과 같다.

```
void zero(int *val) { *val = 0; }
```

함수호출은 x의 주소를 함수 zero에 넘기는것으로 처리된다.

```
int x = 5;          // Object x has value 5
zero( &x );         // Object x will have value 0
```

C++를 리용하여 참조에 의한 호출을 넘기는데는 이 표현형식이 더 좋다. 그러나 많은 서고들은 C와 C++ 두가지로 처리되도록 서술되는데 변수들은 C표현형식을 리용하여 넘긴다. 따라서 이러한 표현형식을 리용하는것이 좋다. 다른 경우에는 그것을 리용하지 않는다.

C형의 배렬과 문자렬

C++언어는 C형으로 만든 배렬형을 제공한다. 10개의 옹근수들로 된 배렬 arr를 선언하기 위하여 다음과 같이 서술한다.

```
int    arrl[10];
```

일반적으로 arr1은 1차클라스배렬형이라기보다는 10개의 옹근수들을 기억시키는데 충분히 큰 기억기에 대한 지적자이다. 따라서 배렬들에 =기호를 적용하는것은 배렬전체보다도 2개의 지적자값만을 복사하기 위한 시도인데 우의 선언에서 arr1은 변하지 않는 지적자이므로 그러한 복사는 허용될수 없다.

배렬 arr1이 함수에로 넘겨 질 때에는 다만 지적자의 값만 넘겨 진다. 즉 배렬의 크기에 대한 정보는 잃어 버린다. 따라서 그 크기는 추가적인 파라메터로 넘겨 진다. 여기서는 size를 알수 없으므로 검사하려는 첨수범위가 없다.

그러나 우의 선언에서 배렬의 크기는 번역할 때 알수 있어야 한다. 배렬의 크기 10은 변수에 의해서 재배치될수 없다. 만일 배렬의 크기를 모르면 지적자를 명백하게 선언하고 기억기를 new[ ]로 할당하여야 한다. 실례로

```
int * arr2 = new int[ n ];
```

에서 arr2는 상수지적자를 쓰지 않은 arr1과 류사하게 작용한다. 따라서 그것은 큰 기억블로크에 대한 지적자로 될수 있다. 그러나 기억기가 동적으로 할당되므로 어떤 시점에서는 그것이 다음과 같이 delete[ ]로 해제되여야 한다.

```
delete[ ] arr2;
```

그렇게 하지 않으면 기억기류실이 발생하게 되는데 만일 배렬이 크다면 그 류실량은 대단하다.

C로 표현된 문자렬은 문자들의 배렬로 실현된다. 문자렬의 길이를 파라메터로 넘기는것을 피하기 위하여 문자렬의 론리적인 끝을 표시하는 문자로서 특별한 빈문자 '\0'이 리용된다. 문자렬들은 strcpy로써 복사되고 strcmp로 비교되며 문자렬들의 길이는 strlen에 의해 결정된다. 개개의 문자들은 배렬첨수연산자에 의해 호출된다. 이 문자렬들에는 어려운 기억기관리문제를 비롯한 배렬과 관련된 문제들이 모두 포함된다. 이것은 그 문자렬을 복사할 때 목적배렬이 결과를 넣는데 아주 충분하다는 가정에 의하여 발생한다. 만일 그렇지 않을 때에는 NULL문자의 왼쪽에 빈 자리가 없으므로 오유수정작업이 어렵게 된다.

부록 B에서는 vector클라스와 string클라스를 서술하는데 그것은 C에 내장된 배렬표현과 문자렬의 성질을 숨기는 방법으로 실현하였다. 이 클라스들을 통하여 사용자는 C에서의 배렬과 문자렬을 조작하는 방법을 익히게 된다. 대부분의 경우 부록 B에 있는 vector와 string클라스(또한 C++체계가 현재 있으면 C++서고에 정의된것)를 리용하는것이 좋지만 C와 C++로 처리되도록 설계된 서고루틴들을 가지고 작업할 때에는 사용자들은 C표현을 리용하는데 관심을 가지게 된다. 또한 속도를 개선하여야 하는 코드부분에서는 C표현을 리용하는것이 때로는 더 효과적일 때도 있다(그러나 이것은 드물다.).

제6절. 형판

배렬에서 가장 큰 항목을 찾는 문제를 생각해 보자. 간단한 알고리듬은 순차적으로 탐색하는것인데 매개 항목을 순차적으로 검사하여 최대값의 위치를 기억한다. 많은 대표적인 알고리듬들가운데서 순차탐색알고리듬은 형에 무관계하다. 형독립에 의하여 이 알고리듬의 론리가 배렬에서 보관되는 항목들의 형에 관계되지 않는다는것을 의미한다. 이와 같은 론리는 옹근수배렬, 실수배렬 그리고 비교가 정확하게 정의될수 있는 어떤 자료형에 대해서도 성립한다.

여기서는 형독립인 알고리듬들과 자료구조들을 서술한다. 형독립인 알고리듬이나 자료구조에 대한 C++코드를 서술할 때 그것을 각이한 형으로 기록하는것보다 한가지로 서술하는것이 더 좋다.

이 절에서는 형독립인 알고리듬(또는 같은 류형의 알고리듬으로 알려 진)들을 template를 리용하여 C++로 서술하는 방법을 고찰한다. 먼저 함수형판을 고찰하고 다음 클라스형판을 설명한다.

1. 함수형판

함수형판들은 일반적으로 서술하기가 아주 쉽다. 함수형판은 실제함수가 아니지만

대신에 함수로 전환될수 있는 하나의 견본이다. 프로그람 1-16은 프로그람 1-11에서 보여 준 string루틴과 사실상 같은 함수형판 findMax를 보여 주었다. template선언을 포함하는 행은 Comparable이 형판인수이며 그것은 임의의 형에 의해서 함수로 재배치될수 있다는것을 의미한다. 즉. 실례로 findMax라고 하는 함수를 vector<string>을 파라메터로 만든다면 함수는 string으로 Comparable을 치환하여 얻는다.

```
        /**
        * Return the maximum item in array a.
        * Assumes a.size( ) > 0.
        * Comparable objects must provide operator< and operator=
        */
        template <c1ass Comparab1e>
        const Comparable & findMax( const vector<Comparab1e> & a )
        {
/* 1*/      int maxIndex = 0;
/* 2*/      for( int i = 1; i < a.size( ); i++ )
/* 3*/          if( a[ maxIndex ] < a[ i ] )
/* 4*/                  maxIndex = 1;
/* 5*/      return a[ maxIndex ];
        }
```

프로그람 1-16. findMax 함수형판

프로그람 1-17은 함수형판이 사용자가 필요한것만큼 자동적으로 확장된다는것을 보여 준다. 매개 새로운 형에 대한 확장은 추가적인 코드를 생성한다. 이것을 코드증가라고 하는데 그것은 큰 대상과제들에서 발생한다. 또한 findMax(v4)호출은 번역시간오유를 발생시킨다. 이것은 Comparable이 Int Cell에 의해 재배치될 때 그림 1-17에서 3행이 허용될수 없기때문이다. 즉 여기에는 IntCell에 대하여 <함수가 정의되여 있지 않다. 따라서 그것은 인수가 형판인수(들)로 구성된다는것을 설명하는 주해들을 포함하는것(어떤 형판에 대한 우선권)이 일반적이다. 이것은 구축자의 종류가 여러가지라는것을 의미한다. 또한 operator<가 두개의 char*에 대한 지적자값들을 비교하기때문에 findMax는 C표현법으로 문자렬을 처리하지 않는다.

형판인수들은 어떤 클라스형으로 가정할수 있기때문에 파라메터넘기기와 되돌림값넘기기변환이 결정되면 형판인수들이 기본형이 아니라는것을 가정하여야 한다. 그것이 상수참조에 의해서 값이 되돌려 지는 리유이다.

함수형판들을 처리하는데는 여러가지 규칙들이 있다. 형판이 파라메터들의 정확한 쌍을 제공할수 없을 때에는 많은 문제점들이 발생하지만 그것들을 얼마든지 없앨수 있다(implicit형변환에 의해서). 거기에는 모호성을 해결하기 위한 방법들과 아주 복잡한 규칙들이 있어야 한다. 만일 형판과 비형판이 있으면 두개가운데서 비형판이 우선권을 가

진다. 또한 만일 접근쌍들이 같이 금지되면 그때 코드는 비법으로 되고 번역기는 모호성을 선언한다는것에 주의하여야 한다.

```
int main( )
{
    vector<int>         v1(37);
    vector<double>      v2(40);
    vector<string>      v3(80);
    vector<IntCell>     v4(75);
        //Additional code to fill in the vectors
    cout<<findMax(v1)<<endl;   //OK:Comparable = int
    cout<<findMax(v2)<<endl;   //OK:Comparable = double
    cout<<findMax(v3)<<endl;   //OK:Comparable = string
    cout<<findMax(v4)<<endl;   //Ilgal operator<undefined
    return 0;
}
```

프로그람 1-17. findMax 함수형판의 리용

중요한것은 많은 번역기들에서 함수형판들은 서로 개별적으로 번역될수 없다는 사실이다. 일반적으로 그것들의 완전한 정의는 .h파일들에 배치되는데 그것을 필요로 하는 모든 프로그람에 포함된다.

2. 클라스형판

가장 간단한 판본에서 클라스형판은 함수형판과 매우 류사하게 동작한다. 프로그람 1-18에 MemoryCell형판을 보여 주었다. MemoryCell형판은 IntCell클라스와 같은데 임의의 Object에 대하여 동작한다. 이 Object는 령파라메터구축자, 복사구축자, 복사대입연산자를 가지고 있다.

이 Object는 상수참조에 의해 넘겨 진다는데 대하여 주의하여야 한다. 또한 0이 쓸모 있는 Object가 아니기때문에 이 구축자에 대한 암시적인 파라메터는 0이 아니라는데 대하여서도 주의하여야 한다. 대신에 암시적인 파라메터는 그것의 령파라메터구축자로 Object를 구축한 결과이다.

```
 /**
 * A class for simulating a memory cell.
 */
template <c1ass Object>
class MemoryCe11
{
     public:
       explicit MemoryCe11 ( const Object & initialValue = Object( ) )
                 : storedVa1ue( initialValue ) { }
```

```
        const Object& read( ) const
            { return stordeValue; }
        void write( const Object & x )
            { storedValue = x; }
    private:
        Object storedValue;
};
```

프로그람 1-18. 분리되지 않는 MemoryCell 형판클라스

프로그람 1-19는 MemoryCell이 표준형들과 클라스형들에 대한 객체를 보관하기 위하여 어떻게 리용되는가를 보여 준다. MemoryCell은 클라스가 아니다. 그것은 다만 클라스형판이다. MemoryCell<int>와 MemoryCell<string>은 실제적인 클라스이다.

만일 클라스형판들을 개개의 단위로 실현하려면 아주 간단한 문법적형식이 필요하다. 사실상 많은 클라스형판들은 이 방법으로 실현된다. 왜냐하면 개별적으로 편집된 형판들은 대부분의 환경에서 잘 동작하지 않기때문이다. 그래서 많은 경우에 실현된 클라스는 모두 .h파일에 배치된다. STL의 일반적인 실현은 이 방법론에 따른다.

```
int main( )
{
        MemoryCell<int> ml;
        MemoryCell<string> m2( "hello" );
        ml.write( 37 );
        m2.write( m2.read( ) + “world" );
        cout << ml.read( ) << endl << m2.read( ) << endl;
        return 0;
}
```

프로그람 1-19. MemoryCell함수형판클라스를 리용하는 프로그람

```
/**
 *   A class for simulating a memory cell.
 */
template <c1ass Object>
class MemoryCell
{
   public:
        explicit MemoryCell( const Object& initial Value = Object( ) );
        const Object & read( ) const;
        void write( const Object & x );
   private:
       Object storedValue;
 };
```

프로그람 1-20. MemoryCell형판클라스의 대면부

그러나 결국 개별적인 편집이 가능하며 그것은 오히려 클라스들에서 처리하는 방법대로 클라스형판의 대면부와 실현부를 분리하는것이 더 좋다. 그러나 이것은 어떤 문법적인 형태를 추가하여야 한다.

프로그람 1-20은 형판클라스에 대한 대면부를 보여 준다. 물론 이 부분은 이미 고찰한 전체 클라스의 한부분이므로 매우 간단하다.

실현부에는 함수형판들이 집합되여 있다. 이것은 매개 함수가 형판을 의미하는 문형을 포함하여야 하며 령역해결연산자를 쓸 때 클라스의 이름은 형판인수와 함께 제시되여야 한다는것을 의미한다. 따라서 프로그람 1-21에서 클라스의 이름은 Memory Cell<Object>이다. 문법이 충분히 표현되였고 그것은 상당히 실제적이라는것을 알수 있다. 실례로 대면부에서 operator=를 정의하는 경우에 특별한 문형을 요구하지 않는다. 그것은 다음과 같이 실현한다.

```
template <class Object>
const MemoryCell<Object> &
MemoryCell<Object>::operator=( const MemoryCell<Object>& rhs)
{
    if this != &rhs )
        storedValue = rhs.storedValue;
    return *this;
}
```

일반적으로 보다 복잡한 함수들의 선언부는 한행에 서술하기에는 너무 길어서 여러개의 행을 요구할 때가 있다.

```
# inc1ude "MemoryCell.h"
/**
 *Construct the MemoryCell with initialValue.
 */
template <c1ass Object>
MemoryCell <Object>::MemoryCell( const Object &   initialValue )
            : storedValue( initialValue )
{
}

/**
* Return the stored value.
*/
template <c1ass Object>
const Object& MemoryCell<Object>::read( ) const
{
      return storedValue;
}
```

```
/**
 * Store x.
 */
template <c1ass Object>
void MemoryCell<Object>::write( const Object & x )
{
        storedValue = x;
}
```

프로그람 1-21. MemoryCell형판클라스의 실현부

설사 클라스형판의 대면부와 실현부가 분리되여 있어도 일부 번역기들은 개별적인 편집을 자동적으로 정확히 처리한다. 가장 간단한 해결방법은 실현부를 끌어 들이기 위하여 대면부파일의 앞끝에 #include표시를 추가하는것이다. 이것이 직결(on-line)코드이다. 또 다른 해결방법은 대상과제안에 있는 개개의 .cpp파일과 마찬가지로 매개 형에 명시적인 실체를 추가하는것이다. 이러한 상세한 내용들은 부단히 변하기때문에 정확한 해결방법을 찾기 위해서는 최신문서들을 참고하여야 한다.

3. 객체, Comparable, 실례

이 책에서는 같은 류형으로서 객체와 Comparable을 반복적으로 리용한다. 객체가 파라메터가 없는 구축자, 대입연산자, 복사구축자를 가진다고 가정하자. findMax에서 본 실례와 같이 Comparable은 총체적인 순서를 주기 위하여 리용될수 있는 <연산자의 형태로 추가적인 기능을 가진다.[9]

프로그람 1-22에서는 비교에 요구되는 기능을 실현하는 클라스의 실례와 연산자다중정의를 보여 준다. 연산자다중정의는 내부연산자의 의미를 정의하는데 리용할수 있다. Employee클라스는 name과 salary를 포함하며 salary에 기초하여 <연산자를 정의한다. 더욱 완전한 <연산자가 가능한데 실례로 name자료성원을 리용하여 salary와의 련계를 끊어 버릴수 있다. 또한 Employee클라스는 령파라메터구축자와 대입연산자, 복사구축자도 준다(암시적으로). 따라서 findMax에서 comparable을 충분히 리용할수 있다.

```
Class Employee
{
        public:
             void setVa1ue( const string& n, double s )
```

[9] 제 12 장의 일부 자료구조들에서는 <연산자에 추가하여 ==연산자를 리용한다. 총적인 순서를 주기 위하여 a<b 그리고 a>b 가 둘다 거짓으로 되면 a==b 이다. 따라서 ==연산자의 리용은 간단해서 편리하다.

```
        { name = n; salary = s; }

    void print( ostream& out ) const
      { out << name << " (" << salary << ")"; }
    bool operator< ( const Employee & rhs ) const
      { return salary < rhs.salary; }

    // Other general accessors and mutators, not shown
  private:
    string name;
    double salary;
};

// Define an output operator for Employee
ostream & operator<< ( ostream & out, const Employee& rhs )
{
        rhs.print( out );
        return out;
}

int main( )
{
    vector<Emp1oyee> v( 3 );

    v[ 0 ].setVa1ue( "Bill Clinton", 200000.00 );
    v[ 1 ].setVa1ue( "Bill Gates", 2000000000.00 );
    v[ 2 ].setVa1ue( "Billy the Marlin", 60000.00 );
    cout << f1ndMax( v ) << endl;
    return 0;
}
```

프로그램 1-22. Comparable은 Employee와 같은 클라스형일수 있다.

실제적으로 응용하기 위하여 자료성원들은 public이거나 또는 추가적인 접근자나 변경자를 제공하여야 한다. 프로그램 1-22는 setValue성원함수와 새로운 클라스형에 대한 출력기능을 주기 위하여 널리 쓰이는 표현형식을 보여 준다. 그 표현형식은 ostream파라메터를 가진 print라고 하는 public성원함수를 준다. 그 public성원함수는 클라스의 함수가 아닌 대역적인 operator<<연산자에 의해서 호출되는데 그 연산자는 ostream과 객체를 접수한다.[10]

[10] 이 표현형식에 의한 해결은 print의 론리를 직접 <<연산자를 가지고 실현하는것이다. <<연산자는 클라스성원이 아니기때문에 C++문법에서는 Employee클라스의 friend함수로 만들어야 한다. 이 대책은 대역적인 형판기능으로 된 friend선언을 정확히 결합할수 없는 낡은 번역기들에서는 작업할수 없는 결함을 가지고 있다. 또한 계승을 포함하는 좀 더 복잡한 내용들에서도 정확히 작업할수 없는 결함이 있는데 그것은 이 책의 범위를 벗어 난다.

제7절. 행렬의 리용

제10장에 있는 몇가지 알고리듬들은 2차원배렬을 리용하는데 이 2차원배렬을 일반적으로 행렬이라고 한다. C++서고는 행렬과 관련된 클라스를 주지 않는다. 그러나 요구되는 행렬클라스는 쉽게 만들수 있다. 기본내용은 벡토르의 벡토르를 리용하는것이다. 이러한 수법은 연산자다중정의에 대한 보충적인 지식을 요구한다. 행렬에서는 [ ]연산자 즉 배렬첨수연산자를 정의한다. 행렬클라스를 프로그람 1-23에서 보여 주었다.

```
template <c1ass Object>
class matrix
{
    public:
        matrix( int rows, int cols ) : array( rows )
        {
            for( int i = 0; i < rows; i++ )
                array[ i ].resize( cols );
        }
        const vector<Object>& operator[]( int row ) const
            { return .array[ row ]; }
        vector<Object>& operator[]( int row )
            { return array[ row ]; }
        int numrows( ) const
            { return array.size( ); }
        int numco1s( ) const
            { return numrows( ) > 0 ? array[ 0 ].size( ) : 0; }
        private:
            vector< vector<Object> > array;
};
```

프로그람 1-23. 완전한 matrix클라스

1. 자료성원, 구축자, 기본접근자

행렬은 vector<Object>의 vector로 선언되는 array자료성원으로 표현된다. 구축자는 먼저 0파라메터구축자로 구축된 매개 vector<Object>형의 rows항목들을 가지고 array를 구축한다. 따라서 rows의 길이가 0인 Object의 벡토르를 가진다.

그다음에 구축자의 본체가 입력되고 매개 행은 cols렬들을 가지고 크기가 수정된다. 따라서 구축자가 2차원배렬로 완성된다. 그다음에 numrows와 numcols접근자들은 보여준바와 같이 쉽게 실현된다.

2. 첨수연산자 [ ]

operator[ ] 연산자는 matrix m이 있을 때 m[i]가 matrix m의 i번째 행에 대응되는 벡토르를 되돌린다. 이렇게 하면 m[i][j]는 표준으로 벡토르첨수연산자를 리용하여 vector m[i]의 j번째 위치에 있는 입구점이 주어 진다. 따라서 matrix operator[ ]는 vector<Object>대신에 Object가 아닌 값을 되돌린다.

operator[ ]는 vector<Object>형의 실체를 되돌려야 한다. 행렬에서 값에 의한 되돌림, 참조에 의한 되돌림, 상수참조에 의한 되돌림을 리용할수 있는가를 보자. 되돌리는 실체가 크기때문에 값에 의한 되돌림은 즉시 무시되지만 호출후에 존재한다는것은 담보된다. 따라서 참조와 상수참조에 의한 되돌림을 리용한다. 다음의 산법을 고찰하자(가명을 사용하거나 크기가 잘못될수도 있는데 그것은 여기서의 고찰이 그 알고리듬을 실행시키자는것이 아니기때문이다.).

```
void copy( const matrix<int>& from, matrix<int>& to )
{
        for( int i = 0; i < to.numrows( ); i++ )
                to[ i ] = from[ i ];
}
```

copy함수에서는 행렬 from의 매개 행을 행렬 to의 대응하는 행에로 복사한다. 명백한것은 operator[ ]가 상수참조를 되돌리면 그때 to[i]는 대입문의 왼변에 대입되지 않는다. 따라서 opertor[ ]는 참조를 되돌려야 한다. 그러나 그렇게 되면 그때 from[i]=to[i]와 같은 표현이 번역되는데 그것은 지어 from이 상수행렬이라고 하더라도 from[i]는 상수벡토르가 아니기때문이다. 그것은 훌륭한 설계를 할수 없게 한다. 이렇게 실제로 operator[ ]가 from에 대한 상수참조를 되돌리는것이 필요하지만 to에는 명백히 참조가 되돌려 진다. 다시말하면 operator[ ]의 두개 준위가 필요한데 그것은 연산들의 되돌리는 형에 따라 다르다. 이것은 허용될수 없지만 한가지 방도가 있다. const가 붙은 성원함수(그 함수가 접근자이든 변경자이든)가 signature의 부분이므로 사용자는 상수참조를 되돌리는 operator[ ]의 접근자준위나 간단한 참조를 되돌리는 변경자준위를 가진다. 그러면 모든것이 원만히 처리되는데 이 과정을 프로그람 1-23에서 보여 주었다.

3. 해체자, 복사대입연산자, 복사구축자

이것들은 모두 vector가 관리하기때문에 자동적으로 처리된다. 따라서 이것들은 모두 matrix클라스의 완전한 기능에 필요한 코드이다.

요약

이 장에서는 이 책의 기초로 되는 지식을 주었다. 입력되는 자료량이 큰 경우 어떤 알고리듬이 주어 지면 그 알고리듬이 좋은가를 결정하는 중요한 기준이 있다(물론 정확성이 가장 중요하다.). 무엇보다도 속도가 관계된다.

어떤 체계에서 하나의 문제가 빠르게 수행되는것도 다른 문제나 다른 도구에서는 속도가 떠질수도 있다. 다음 장에서부터는 이에 대하여 고찰하며 형식화된 모형을 세우기 위하여 이 장에서 론의된 수학적인 방법들을 리용한다.

련습문제

1-1. 선택문제를 풀기 위한 프로그람을 작성하시오. $k=N/2$인 때 변수값 N에 대한 프로그람의 실행시간을 보여 주는 표를 만드시오.

1-2. 단어맞추기문제를 푸는 프로그람을 작성하시오.

1-3. printDigit를 리용하여 임의의 double형의 부수를 출력하는 함수를 작성하시오.

1-4. C++에서는 다음과 같은 형태의 지령을 볼수 있다.

```
#include  filename
```

이것은 파일이름을 읽고 include문의 위치에 그 내용을 삽입한다. include문들은 해당 코드의 앞부분에 놓이게 되는데 다시말하면 파일 filename은 그자체가 include문에 의해 포함되게 되지만 파일은 어떤 련결에 자기자체를 포함시킬수 없다. 파일을 읽어 들이고 include문으로 수정된 파일을 출력하는 프로그람을 작성하시오.

1-5. N을 2진수로 표현하고 1의 자리의 수자를 되돌리는 재귀함수를 작성하시오. 여기에서 N이 주어 지면 $N/2$에 1을 더한 값이 1의 자리수자로 된다는 사실을 리용하시오.

1-6. 다음의 선언문들을 가진 루틴을 작성하시오.

```
void permute(const string& str);
void permute(const string& str, int low, int high);
```

첫번째 루틴은 두번째 루틴을 호출하여 string str에 있는 문자들을 모두 변경하여 출력한다. 만일 str가 "abc"이면 abc, acb, bac, bca, cab, cba인 문자렬들이 출력된다. 두번째 루틴에서는 재귀방법을 리용하시오.

1-7. 다음의 규칙들을 증명하시오.

ㄱ. $\log X < X$이면 모든 X에 대하여 $X > 0$이다.

ㄴ. $\log(A^B) = B\log A$

1-8. 다음의 합을 구하시오.

ㄱ. $\sum_{i=0}^{\infty} \frac{1}{4^i}$ ㄴ. $\sum_{i=0}^{\infty} \frac{i}{4^i}$ *ㄷ. $\sum_{i=0}^{\infty} \frac{i^2}{4^i}$ **ㄹ. $\sum_{i=0}^{\infty} \frac{i^N}{4^i}$

1-9. 다음의 식을 계산하시오.

$$\sum_{i=\lfloor N/2 \rfloor}^{N} \frac{1}{i}$$

*1-10. $2^{100} (\bmod 5)$의 값은 얼마인가?

1-11. F_i는 제1장 제2절에서 정의된 피보나치수렬이다. 다음의 식을 증명하시오.

ㄱ. $\sum_{i=1}^{N-2} F_i = F_N - 2$ ㄴ. $F_N < \phi^N$, $\phi = (1+\sqrt{5})/2$

**ㄷ. F_N에 대한 닫긴형태의 정확한 표현을 주시오.

1-12. 다음의 식을 증명하시오.

ㄱ. $\sum_{i=1}^{N} (2i-1) = N^2$ ㄴ. $\sum_{i=1}^{N} i^3 = (\sum_{i=1}^{N} i)^2$

1-13. 모임의 현재크기를 가지고 Object들의 모임(배렬)들을 보관하는 클라스형판 Collection을 설계하시오. public성원함수들로는 isEmpty, makeEmpty, insert, remove, isPresent가 있다. isPresent(x)는 x와 같은 Object가 모임에 존재하면 true를 되돌린다.

1-14. 모임의 현재크기를 가지고 Comparable들의 모임(배렬로)을 보관하는 클라스형판 OrderedCollection을 설계하시오. public함수들로는 isEmpty, makeEmpty, insert, remove, findMin, findMax가 있다. findMin과 findMax는 각각 모임에 있는 Comparable에서 가장 작거나 가장 큰것에 대한 참조를 되돌린다. 만일 이 연산들이 빈 모임에서 실현되면 어떻게 되겠는가를 설명하시오.

1-15. matrix클라스에서 resize성원함수와 파라메터가 없는 구축자를 추가하시오.

참고문헌

이 장에서 취급한 수학적인 내용들을 담은 좋은 책들은 많다. 그중 일부가 [1], [2], [3], [9], [14], [16]에 있다. 참고문헌 [9]는 알고리듬의 분석을 위하여 필요하다. 그것은 이 책전체에 걸쳐 렬거되는 세가지 계렬의 책들중에서 첫번째 계렬에 속한다. 좀 더 개선된 내용들은 [6]에 있다.

이 책에서는 전반적으로 C++에 대한 지식을 전제로 하고 있다. 책 [15]는 대체로 C++의 최종설계표준을 서술하는데 그것은 C++의 초기설계자가 쓴것으로서 대부분 저작권을 가지고 있다. 또한 다른 표준적인 참고문헌은 [10]이다. 그리고 C++에서 개선된 내용들은 [5]에서 서술되였다. 두개의 계렬 [11,12]는 C++에 대한 많은 문제점들에 대하여 서술하였다. 부록 A에 소개된 표준형판서고는 [13]에 서술되였다. 제1장 제4절~제7절의 내용들은 이 책에서 쓰이는 특징적인 문제들을 고찰하였다. 또한 지적자와 재귀에 대해서도 서술하였다(이 장에서 취급한 재귀에 대한 내용은 단지 속성적인 측면에 불과하다.). 책의 전반에 걸쳐 적당한 위치에서 그것들의 쓰임에 대한 내용을 주기 위해 노력하였다. 이 내용들을 잘 모르는 독자들은 [17]이나 또는 다른 좋은 프로그람작성교재들을 참고하기 바란다.

일반적인 프로그람작성방법은 여러 책들에 서술되여 있다. 그중 일부가 [4], [7], [8]이다.

1. M. O. Albertson and J. P. Hutchinson, *Discrete Mathematics with Algorithms,* John Wiley & Sons, New York, 1988.
2. Z. Bavel, *Math Companion for Computer Science.* Reston Publishing Co., Reston, Va., 1982.
3. R. A. Brualdi, *lntroductory Combinatorics,* North-Holland, New York, 1977.
4. E. W. Dijkstra, A *Discipline of Programming,* Prentice Hall. Englewood Cliffs, N.J., 1976.
5. B. Eckel, *Thinking in* C++ , Pretice Hall, Englewood Cliffs, N.J., 1995.
6. R. L. Graham, D. E. Knuth, and O. Patashnik, *Concrete Mathematics,* Addison-Wesley, Reading, Mass., 1989.
7. D.Gries *The Science of programming,* Springer-verlag, New York, 1981
8. B. W. Kernighan and P. J. Plauger, *The Elements of Programming Style,* 2d cd ,McGraw-Hill, New York. 1978.
9. D. E. Knuth, *The Art of Computer Programming, Vol. 1: Fundamental Algorithms,* 3d ed., Addison-Wesley, Reading, Mass., 1997.
10. S. B. Lippman and J. Lajoie, C++ *Primer,*3d ed., Addison-Wesley, Reading, Mass., 1998.
11. S. Meyers, 50 *Specific Ways to Improve Your Programs and Designs,* 2d ed., Addison-Wesley, Reading, Mass., 1998.
12. S. Meyers, *More Effective* C++: 35 *New Ways to Improve Your Programs and Designs,* Addison-Wesley, Reading, Mass., 1996.
13. D. R. Musser and A. Saini, *STL Tutorial and Reference Guide:* C++ *Programming with the Standard Template Library,* Addison-Wesley, Reading, Mass., 1996.
14. F. S. Roberts, *Applied Combinatorics,* Prentice Hall, Englewood Cliffs, N.J., 1984.
15. B. Struustrop, *The C++ Programming Language,* 3d ed., Addison-Wesley, Reading, Mass., 1997.
16. A. Tucker, *Applied Combinatorics,* 2d ed.,John Wiley & Sons, New York, 1984.
17. M. A. Weiss, *Algorithms, Data Structures, and Problem Solving with* C++, Addison-Wesley, Reading, Mass., 1996.

제2장. 알고리듬분석

알고리듬은 문제를 풀기 위하여 따라 가야 할 명백하게 지정된 간단한 지령들의 모임이다. 일단 어떤 문제를 풀기 위하여 하나의 알고리듬이 주어 지고 정확하다는것이 판단되면 그다음 중요한 단계는 알고리듬이 시간이나 공간과 같은 자원을 얼마나 많이 요구하는가를 결정하는것이다. 어떤 문제를 푸는데 1년이 걸리는 알고리듬은 사실상 리용할수 없다. 마찬가지로 수기가바이트의 주기억기를 요구하는 알고리듬도 현재로서는 대부분의 기계들에서는 쓸모 없다.

이 장에서는 다음의 내용들을 고찰한다.

- 프로그람실행에 요구되는 시간을 평가하는 방법
- 며칠 또는 몇년으로부터 1초이하로 프로그람의 실행시간을 단축하는 방법
- 재귀에 대한 서툰 리용의 결과
- 어떤 수의 제곱과 두 수들의 최대공통약수를 계산하는 매우 효과적인 알고리듬들

제1절. 수학지식

알고리듬의 자원리용을 평가하는데 요구되는 분석들은 일반적으로 리론적인 문제들이며 따라서 형식적인 틀거리가 요구된다. 여기서는 몇가지 수학적정의로부터 시작한다. 이 책의 전반에 걸쳐 대체로 다음과 같은 4가지 정의를 리용한다.

정의: $N \geqq n_0$인 때 $T(N) \leqq cf(N)$을 만족하는 정의상수 c와 n_0이 있으면 $T(N)=O(f(N))$이다.

정의: $N \geqq n_0$인 때 $T(N) \geqq cg(N)$을 만족하는 정의상수 c와 n_0이 있으면 $T(N)=\Omega(g(N))$이다.

정의: 만일 $T(N)=O(h(N))$이고 $T(N)=\Omega(h(N))$이면 $T(N)=\Theta(h(N))$이다.

정의: $T(N)=O(p(N))$이고 $T(N) \neq \Theta(p(N))$이면 $T(N)=O(p(N))$이다.

이 정의들에 대한 개념은 함수들의 상대적인 순서를 결정하는것이다. 어떤 두함수가 주어 지면 한 함수가 다른 함수보다 더 작아지게 되는 경우들도 있는데 그렇다고 하여 가령 $f(N)<g(N)$이라고 주장하는것은 리치에 맞지 않는다. 따라서 여기에서는 그의 **상대적증가률**(*relative rate of growth*)을 비교한다. 상대적증가률을 알고리듬분석에 적용하면 이것이 왜 중요한 측도로 되는가를 알게 된다.

N이 작을 때에는 $1,000N$이 N^2보다 더 크지만 N^2이 더 빨리 증가하므로 결국

N^2이 더 큰 함수로 될것이다. 이 경우에 크기가 변하는 점은 N=1,000인 때이다. 첫번째 정의는 $c \cdot f(N)$이 항상 적어도 $T(N)$만큼 커지게 되는 어떤 값 n_0이 있다는것을 의미하며 따라서 상수인자들을 무시하면 $f(N)$이 적어도 $T(N)$만큼 크다는것을 말해 준다. 이 경우에는 $T(N)$=1,000N, $f(N)=N^2$, n_0=1000, c=1이다. 또한 n_0=10, c=100을 리용할수도 있다. 따라서 $1000N=O(N^2)$이라고 할수 있다(N제곱순차수). 이 표기를 **큰O표기법**(*Big-Oh notation*)이라고 한다. 대체로 《…순차수》라고 부르지 않고 《큰O…》라고 부른다.

만일 증가률을 비교하는데 일반적으로 써오던 부등식기호를 리용한다면 첫번째 정의는 $T(N)$의 증가률이 $f(N)$의 증가률보다 작거나 같다(≦)는것을 의미한다. 두번째 정의 $T(N)=\Omega(g(N))$(《오메가》라고 부른다)은 $T(N)$의 증가률이 $g(N)$보다 크거나 같다(≧)는것을 의미한다. 세번째 정의 $T(N)=\Theta(h(N))$(《시타》라고 부른다)는 $T(N)$의 증가률이 $h(N)$의 증가률과 같다(=)는것을 의미한다. 마지막 정의 $T(N)=o(p(N))$(《작은 오우》라고 부른다)는 $T(N)$의 증가률이 $p(N)$의 증가률보다 작다(<)는것을 의미한다. 이것은 큰O가 그 증가률들이 같아 질 가능성을 주므로 큰O와 차이난다.

어떤 함수가 $T(N)=O(f(N))$이라는것을 증명하기 위하여 보통 이 정의들을 그대로 적용하지 않고 대신에 경험적으로 알려 진 결과들을 리용한다. 일반적으로 이것은 증명(또는 가정이 타당치 않다는것을 결정하는것)이 매우 단순한 계산으로 되며 특별한 경우를 제외하고는 제곱연산을 포함하지 않는다는것을 의미한다(알고리듬분석에서 생기는것과는 다르다).

$T(N)=O(f(N))$이면 함수 $T(N)$은 $f(N)$보다 빠르지 않은 비률로 증가한다는것이 담보되며 따라서 $f(N)$은 $T(N)$에 대한 웃한계이다. 이것은 또한 $f(N)=\Omega(T(N))$라는것을 의미하므로 $T(N)$을 $f(N)$에 대한 아래한계이라고 한다.

실례로 N^3은 N^2보다 더 빨리 증가하며 따라서 $N^2=O(N^3)$ 또는 $N^3=\Omega(N^2)$이라고 할수 있다. $f(N)=N^2$과 $g(N)=2N^2$은 같은 비률로 증가하므로 $f(N)=O(g(N))$과 $f(N)=\Omega(g(N))$은 둘 다 성립한다. 두 함수가 같은 비률로 증가할 때 이것을 $\Theta()$로 표시하는가 아닌가 하는것은 개별적인 문맥에 의존할수 있다. 만일 $g(N)=2N^2$이면 $g(N)=O(N^4)$, $g(N)=O(N^3)$, $g(N)=O(N^2)$은 모두 학술적으로 정확하지만 마지막 선택이 가장 좋은 결과라는것을 직관적으로 알수 있다. $g(N)=\Theta(N^2)$로 표기하면 $g(N)=O(N^2)$일뿐아니라 그 결과가 괜찮게 좋은(엄밀한)것이라는것을 의미한다.

여기서 반드시 알아야 할 중요한 문제들은 다음과 같다.

규칙 1.

만일 $T_1(N) = O(f(N))$이고 $T_2(N) = O(g(N))$이면

(ㄱ) $T_1(N) + T_2(N) = \max(O(f(N)), O(g(N)))$,

(ㄴ) $T_1(N) * T_2(N) = O(f(N) * g(N))$이다.

규칙 2.

만일 $T(N)$이 k차다항식이면 $T(N) = \Theta(N^k)$이다.

규칙 3.

어떤 상수 k에 대하여 $\log^k N = O(N)$이다. 이것은 로그식이 대단히 천천히 증가한다는것을 의미한다.

이 정보들은 대부분의 일반함수들을 증가률에 따라 정렬하는데 충분하다(표 2-1을 보시오.).

표 2-1. 전형적인 증가률

함 수	이 름
c	상 수
$\log N$	로 그
$\log^2 N$	로그두제곱
N	선 형
$N\log N$	
N^2	2차원
N^3	3차원
2^N	지 수

몇가지 문제점들을 차례로 보자. 첫째로, 큰O의 내부에 상수들이나 **낮은 차수항**들을 포함하는것은 대단히 나쁜 표현법이다. $T(N) = O(2N^2)$ 또는 $T(N) = O(N^2+N)$라고 표현하지 말아야 한다. 두 경우에 정확한 형태는 $T(N) = O(N^2)$이다. 이것은 큰O결과를 요구하는 어떠한 분석에서나 모든 정렬들의 간단한 방법이 있을수 있다는것을 의미한다. 낮은 차수항들은 일반적으로 무시될수 있으며 상수들은 버릴수 있다. 이 경우에 요구되는 정확도는 상당히 낮다.

둘째로, 필요하다면 L'Hopital의 규칙을 리용하여 $\lim_{N\to\infty} f(N)/g(N)$을 비교하면 두 함수 $f(N)$과 $g(N)$의 상대적인 **증가비률**을 결정할수 있다.[11] 그 극한은 4개의 가능한 값을 가질수 있다.

---

[11] L'Hopital의 규칙은 $\lim_{N\to\infty} f(N)=\infty$이고 $\lim_{N\to\infty} g(N)=\infty$이면 $\lim_{N\to\infty} f(N)/g(N)=\lim_{N\to\infty} f'(N)/g'(N)$라는것이다. 여기서 $f'(N)$과 $g'(N)$은 각각 $f(N)$과 $g(N)$의 도함수이다.

- 극한은 0이다. 즉 $f(N)=o(g(N))$이다.
- 극한은 $c \neq 0$이다. 즉 $f(N) = \Theta(g(N))$이다.
- 극한은 ∞이다. 즉 $g(N) = o(f(N))$이다.
- 극한은 발산한다. 즉 관계가 없다(이것은 현재 상황에서는 발생하지 않을것이다).

이 산법을 리용하면 거의 언제나 파잉으로 된다. 보통 $f(N)$과 $g(N)$사이의 관계는 간단한 산수계산으로 유도될수 있다. 실례로 만일 $f(N)=N\log N$이고 $g(N)= N^{1.5}$라면 $f(N)$과 $g(N)$중 어느것이 더 빨리 증가하는가를 결정하기 위해서 실제로 $\log N$과 $N^{0.5}$ 가운데서 어느것이 더 빨리 증가하는가를 결정하여야 한다. 이것은 $\log^2 N$과 N가운데서 어느것이 더 빨리 증가하는가를 결정하는것과 등가이다. N이 로그의 임의의 제곱값보다 더 빨리 증가한다는것은 이미 알고 있는 사실이므로 이 문제는 간단하다. 결국 $g(N)$이 $f(N)$보다 더 빨리 증가한다.

표현상 주의점: $f(N) \leqq O(g(N))$이라고 하는것은 나쁜 표현이다. 그것은 정의에 의하여 부등식이 암시되기때문이다. $f(N) \geqq O(g(N))$이라고 쓰는것도 그리 좋지 못한데 그것은 리치에 맞지 않기때문이다.

일반적인 분석실례로서 인터네트상의 파일을 내리적재하는 문제를 생각해 보자. 처음에 접속을 위한 3초의 지연이 있고 그다음에 1.5K(byte)/s의 속도로 내리적재를 진행한다고 가정한다. 이때 만일 그 파일이 N키로바이트라면 내리적재시간은 식 $T(N)=N/1.5+3$으로 표현된다. 이것은 선형함수이다. 1,500K파일을 내리적재하는데 걸리는 시간(1003s)은 근사적으로(정확하지는 않지만) 750K파일을 내리적재하는데 걸리는 시간(503s)의 두배이다. 이것은 선형함수의 전형이다. 만일 접속속도가 두배로 되면 두 시간들은 다 감소하지만 1,500K파일은 여전히 750K파일보다 내리적재하는 시간이 거의 두배 걸린다. 이것은 선형시간알고리듬의 전형적인 특징이며 상수인자를 무시하고 $T(N)=O(N)$이라고 표현하는 리유이다.(비록 큰시타를 리용하는것이 더 정확하지만 대체로 큰O결과들이 주어 진다.) 또한 이 성질이 모든 알고리듬에 대하여 맞는것은 아니라는 사실에도 주목을 돌려야 한다. 제1장 제1절에서 서술한 첫번째 선택알고리듬의 실행시간은 정렬을 수행하는데 걸리는 시간에 따라 좌우된다. 거품정렬과 같은 간단한 정렬알고리듬에서 입력량이 2배로 되면 실행시간은 4배로 증가한다. 그것은 이 알고리듬이 선형이 아니기때문이다. 앞으로 정렬에 대하여 론의할 때 알게 되겠지만 일반적인 정렬알고리듬들은 $O(N^2)$ 또는 2차원으로 된다.

제2절. 모형

형식적인 틀거리로 알고리듬을 분석하기 위하여서는 계산모형이 필요하다. 이 모형은 지령들이 순차적으로 실행되는 표준콤퓨터에 기초하고 있다. 이 모형은 더하기, 곱하기, 비교, 대입과 같은 표준적인 간단한 지령항목들을 가지지만 실지 콤퓨터와는 달리 그것은 명백히 어떤 간단한 처리를 수행하는 한개 시간단위를 가진다. 편의를 도모하기 의하여 현대적인 콤퓨터와 같은것을 가상하면 론리적으로 이 모형은 고정된 크기(32bit와 같은)의 옹근수들을 가지며 거기에는 하나의 시간단위로 처리할수 없는 행렬전위나 정렬과 같은 특별한 연산들은 명백히 없다. 또한 기억기의 용량도 무한하다고 가정한다.

이 모형은 확실히 일부 약점들을 가지고 있다. 실제적으로 모든 연산들은 꼭 같은 실행시간을 가지지 않는다. 특히 이 모형에서는 더하기가 훨씬 더 빠르다고 하여도 어떤 디스크읽기를 더하기처럼 계수한다. 또한 기억기한계를 무한하다고 가상하면 사용자는 현실적인 문제로 될수 있는(특히 효과적인 알고리듬의 경우) 페지결함에 대하여 걱정하지 않아도 된다.

제3절. 분석내용

분석에서 가장 중요한 자원은 일반적으로 실행시간이다. 여러가지 인자들이 프로그람의 실행시간에 영향을 준다. 번역기와 콤퓨터와 같은 일부 자원들은 명백히 어떤 리론적인 모형의 범위를 벗어 나므로 그것들이 중요하다고 하여도 여기에서 그것들을 취급할수는 없다. 기타 다른 중요한 인자들은 리용되는 알고리듬과 그 알고리듬에 입력하는 자료들이다.

대표적으로 입력량의 크기는 중요하게 고려해야 할 문제이다. 입력의 크기 N에 대하여 어떤 알고리듬을 리용하여 각각 평균경우와 최악의 경우의 실행시간을 표시하는 두 함수 $T_{avg}(N)$과 $T_{worst}(N)$을 정의하자. 이것은 명백히 $T_{avg}(N) \leqq T_{worst}(N)$으로 된다. 만일 한개이상의 입력자료가 있으면 이 함수들은 한개이상의 인수를 가지게 된다.

때로는 가장 좋은 경우에 실행되는 알고리듬이 분석되기도 한다. 그러나 이것은 일반적인 특징을 나타내지 못하므로 흥미가 적다. 평균경우의 실행은 흔히 일반적인 특징을 나타내며 최악의 경우의 실행은 어떤 가능한 입력에 대한 실행의 담보성을 나타낸다. 또한 이 장의 전반에서 C++코드를 분석한다 해도 이 한계값들은 실제로 프로그람들이 아니라 알고리듬들의 한계값이다. 프로그람들은 알고리듬에 대한 어떤 프로그람작성언어의 구체적인 실현인데 프로그람작성언어의 상세한 내용들은 항상 큰O결과에 영향을 주지 않는다. 만일 프로그람이 알고리듬분석에서 제기한 값보다 훨씬 더 천천히 실행되면 거기에는 비능률적인 실현이 있게 된다. 이러한것은 C++에서 배렬이 참조를 가지고 넘겨 지지 않고 그 전체를 무의식적으로 복사할 때 발생할수 있다. 이에 대한 또 하나의 아주 미묘한 실례는 제12장 제7절의 마지막 2개 구절에 있다. 이와 같이 앞으로는 프로그람이 아니라 알고리듬을 분석하게 된다.

일반적으로 알고리듬분석에서 요구되는 량은 다르게 지적되지 않는 한 최악의 경우의 시간이다. 그 리유는 최악의 경우의 시간이 평균경우의 분석에서는 담보되지 않는 특수하게 틀린 입력을 포함하여 모든 입력에 대한 시간한계를 주기때문이다. 또 한가지 리유는 일반적으로 평균경우의 한계들이 훨씬 더 어렵게 계산되기때문이다. 일부 경우에 《평균》이라는 정의는 그 결과에 영향을 미칠수 있다.(실례로 다음의 문제에서 평균입력은 얼마인가?)

하나의 실례로 다음 절에서는 다음과 같은 문제를 고찰하게 된다.

최대부분순서합문제

주어 진 옹근수(부수도 가능함) A_1, A_2, ⋯, A_N에서 $\sum_{k=i}^{j} A_k$ 의 최대값을 찾으시오(편리상 옹근수들이 모두 부수이면 최대부분순서합은 0으로 한다.).

실례: 입력자료가 -2, 11, -4, 13, -5, -2일 때 그 결과는 20이다($A_2 \sim A_4$).

이 문제를 풀기 위한 많은 알고리듬이 존재하기때문에 이 문제는 아주 흥미있는데 그 알고리듬들은 철저하게 다르다. 이 문제를 풀기 위한 4개의 알고리듬을 고찰하자. 어떤 콤퓨터(어떤 콤퓨터인가 하는것은 중요하지 않다.)상에서 이 알고리듬들에 대한 실행시간을 표 2-2에 보여 주었다.

표 2-2. 최대부분순서합에 대한 여러가지 알고리듬들의 실행시간(s)

알고리듬		**1**	**2**	**3**	**4**
시 간		$O(N^3)$	$O(N^2)$	$C\ NlogN)$	$O(N)$
입력크기	N=10	0.00103	0.00045	0.00066	0.00034
	N=100	0.47015	0.01112	0.00486	0.00063
	N=1,000	448.77	1.1233	0.05843	0.00333
	N=10,000	NA	111.13	0.68631	0.03042
	N=100,000	NA	NA	8.0113	0.29832

다음으로 우의 표에 주어 진 시간들에는 입력자료를 읽어 들이는데 요구되는 시간이 포함되지 않았다. 알고리듬4에서 어떤 디스크로부터 입력자료를 읽어 들이는데 걸리는 시간은 순전히 그 문제를 푸는데 요구되는 시간보다 더 방대할수도 있다. 이것은 많은 효과적인 알고리듬들에서 일반적으로 제기되는 문제이다. 일반적으로 입력자료를 읽어 들이는것은 프로그람의 실행에서 장애로 되는데 일단 자료가 읽어 지기만 하면 그 문제는 고속으로 처리될수 있다. 비능률적인 알고리듬들인 경우에는 이것이 불가능하며 중요한 콤퓨터자원들이 리용되여야 한다. 따라서 가능한껏 알고리듬은 문제풀이에 장애가 되지 않도록 충분히 효과적인것으로 되는것이 중요하다.

그림 2-1에 4개 알고리듬들의 실행시간에 대한 **증가비률**을 보여 주었다. 이 그라프가 N이 10~100범위의 값들만 포함해도 상대적인 증가비률은 뚜렷하게 나타난다. 알고리듬 3의 그라프는 선형적인것처럼 보이지만 곧은자(혹은 종이장)를 리용하면 그것이 선형이 아니라는것을 쉽게 증명할수 있다.

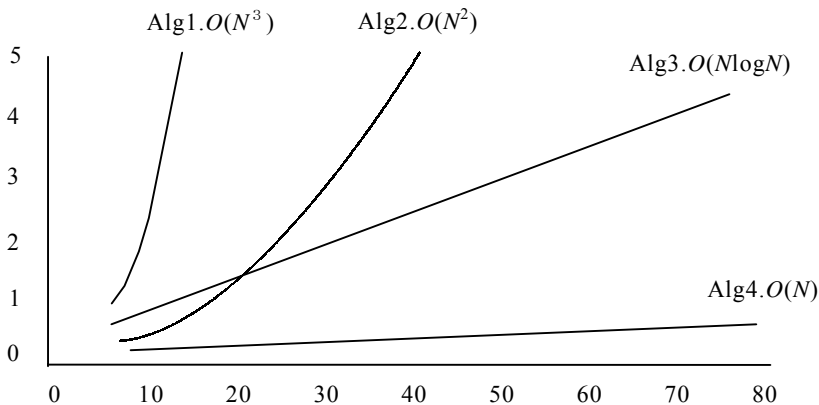



그림 2-1. 여러가지 최대순서합알고리듬들에 대한 그라프(N:ms)

그림 2-2에 더 큰 값들에 대한 실행을 보여 주었다. 그것은 적당히 큰 입력량들에 대해서도 비능률적인 알고리듬들이 얼마나 쓸모 없는가를 잘 설명한다.

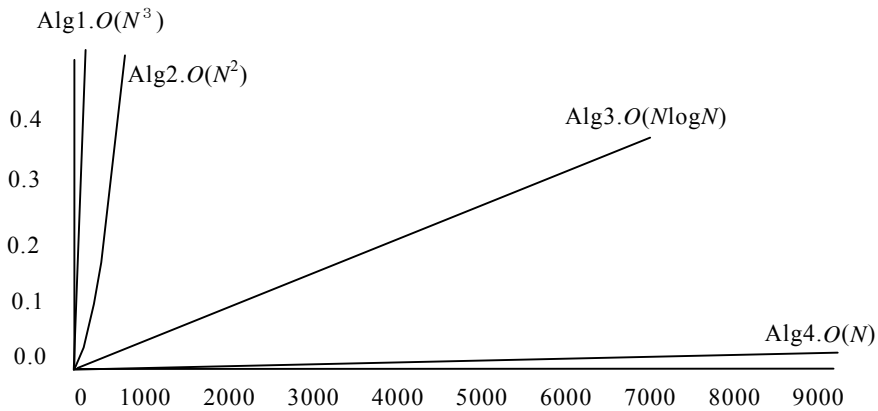



그림 2-2. 여러가지 최대순서합알고리듬들에 대한 그라프(N:s)

제4절. 실행시간계산

프로그람의 실행시간을 평가하는데는 여러가지 방법이 있다. 우에 있는 표는 경험적으로 작성되여 있다. 만일 두 프로그람이 류사한 시간에 수행된다고 예상될 때 더 빠른것을 결정하는 가장 좋은 방법은 그것들을 코드화하고 실행시켜 보는것이다.

일반적으로 여러가지 알고리듬적인 방법들이 있는데 사용자는 사전에 좋지 못한 알고리듬은 없애 버리려고 한다. 그러므로 일반적으로 그 알고리듬들에 대한 어떤 분석이 요구된다. 더우기 분석능력은 보통 효과적인 알고리듬들을 설계할수 있는 통찰력을 준다. 분석은 또한 일반적으로 병목현상들을 정확히 지적할수 있게 하며 알고리듬을 세밀하게 코드화할수 있게 한다.

분석을 간단히 하기 위하여 일반적으로 정밀한 시간단위가 아닌 값을 받아들이게 된다. 따라서 상수들은 없애 버린다. 또한 낮은 차수의 항목들도 없애 버림으로써 본질적으로는 큰O실행시간을 계산한다. 큰O가 웃한계이므로 프로그람의 실행시간을 과소평가하지 않도록 주의하여야 한다. 사실상 제공된 결과는 응당한 시간내에 프로그람이 끝날것이라는 담보를 주어야 한다. 프로그람은 이 보다 더

빨리 완료되여야지 결코 떠지지는 말아야 한다.

1. 간단한 실례

여기에 $\sum_{i=1}^{N} i^3$ 을 계산하기 위한 간단한 프로그람토막이 있다.

```
        int sum( int n )
        {
            int partialSum;
/* 1*/      partialSum = 0;
/* 2*/      for( int = 1; i <= n; i++)
/* 3*/          partialSum += i * i * i;
/* 4*/      return partialSum;
        }
```

이 프로그람토막은 간단히 분석할수 있다. 여기에서 선언들은 시간으로 계산되지 않는다. 1행과 4행은 매개가 하나의 단위로 계산된다. 3행은 매 실행당 4개의 단위로 계산되고(2번의 곱하기와 한번의 더하기, 한번의 대입) 이것이 N번 실행되여 전체적으로 $4N$개의 단위들로 계산된다. 2행은 암시적으로 i를 초기화하고 $i \le N$을 검사하며 i를 증가시키는 시간들을 가진다. 이것들의 전체 값은 한번 초기화되고 N +1번 검사되며 N번 증가하므로 $2N+2$로 계산된다. 함수호출시간과 되돌림시간을 무시하면 총 $6N+4$의 값을 가지므로 이 함수의 실행시간은 $O(N)$임을 알수 있다.

만일 어떤 프로그람을 분석할 때마다 이러한 작업을 모두 수행하여야 한다면 그 작업은 빨리 실행할수 없게 된다. 그러나 그 결과를 큰O로 평가하면 총체적인 결과에 영향을 주지 않고 빨리 계산할수 있다. 우의 실례에서 3행은 명백히 $O(1)$ 명령문(매 실행당)이므로 실행단위가 2이든 3이든 4이든 그것을 명백하게 계산할 필요는 없다. 즉 그것은 문제로 되지 않는다. 1행은 for순환에 비하여 명백히 보잘것 없는 량이므로 여기에 시간을 소비할 필요는 없다. 이로부터 몇가지 일반적인 규칙들을 끌어 낼수 있다.

2. 일반적인 규칙

규칙 1 - for순환

어떤 for순환의 실행시간은 기껏해서 for순환내부의 지령들의 실행시간(검사들을 포함하여)에 반복회수를 곱한 시간이다.

규칙 2 - 다중순환

이것들은 제일 안쪽까지 완전히 분석하여야 한다. 한조의 다중순환의 내부에 있는 지령들의 전체 실행시간은 모든 순환들의 크기의 적으로 곱해 진 지령들의 실행시간이다.

실례로 다음의 프로그람토막은 $O(N^2)$이다.

```
for( i = 0; i < n; i++ )
    for( j = 0; j < n; j++ )
        k++;
```

규칙 3 - 병렬적인 지령

이것들은 응당 더해 져야 한다(그것은 최대값이 총체적인 실행시간값으로 계산된다는것을 의미한다. 제2장 제1절에 있는 규칙 1을 보시오).

실례로 $O(N^2)$처리에 이어 $O(N)$처리를 가지는 다음의 프로그람토막은 역시 $O(N^2)$이다.

```
for( i = 0; i < n; i++ )
        a[ i ] = 0;
for( i = 0; i < n; i++ )
        for( j = 0; j < n; j++ )
            a[ i ] += a[ j ] + i + j;
```

규칙 4 - if/else

프로그람토막

```
if (조건)
        s1
else
        s2
```

에서 if/else지령의 실행시간은 s1과 s2의 실행시간가운데서 더 큰 값을 가지는 실행시간보다 더 크지 않다.

명백히 이것은 어떤 경우에는 과대평가될수 있지만 과소평가되지는 않는다.

다른 규칙들은 명백하지만 안쪽(또는 제일 깊은 부분)으로부터 밖으로 분석하는 기본전략을 리용하여야 한다. 그리고 만일 함수호출이 있게 되면 그것들이 먼저 분석되여야 한다. 재귀함수들이 있으면 여러가지 선택을 할수 있다. 만일 재귀를 실제로 for순환처럼 고찰할수 있다면 이에 대한 분석은 대체로 명백한것이다. 실례로 다음의 프로그람은 사실상 간단한 순환고리와 같은데 시간효과성이 $O(N)$이다.

```
long factorial( int n )
{
        if ( n <= 1 )
          return 1;
        else
           return n*factorial( n – 1 );
}
```

이 실례는 사실상 재귀에 대한 좋지 못한 리용이다. 재귀가 정확하게 리용될 때 그 재귀를 간단한 순환구조로 변환하는것은 어렵다. 이 경우에 그에 대한 분석은 해결되여야 할 재귀관계를 포함하게 된다. 그 과정을 다음의 프로그람으로 고찰하자. 이것도 재귀에 대한 부적합한 리용이다.

```
          long fib(int n)
          {
/* 1*/        if ( n <= 1 )
/* 2*/            return 1;
              else
/* 3*/            return fib( n – 1 ) + fib( n – 2 );
          }
```

얼핏 보기에는 이것이 아주 재치 있는 재귀수법처럼 보인다. 그러나 만일 N에 30정도의 값을 주고 이 프로그람을 실행하여 보면 완전히 비능률적이라는것을 명백히 알수 있다. 이에 대한 분석은 간단하다. 함수 fib(n)에 대한 실행시간을 $T(N)$이라고 하자. N=0이거나 N=1이면 그 실행시간은 어떤 상수값을 가지는데 그것은

1행에 있는 조건을 판정하고 되돌리는 시간이다. 상수들은 문제로 되지 않으므로 $T(0)=T(1)=1$이라고 할수 있다. 다른 N값에 대한 실행시간은 기초과정에 대한 실행시간에 관하여 측정된다. $N>2$에 대한 함수의 실행시간은 1행에서의 상수처리와 3행에서 처리를 더한다. 3행은 한번의 더하기와 2개의 함수호출로 구성된다. 함수호출이 간단한 연산들이 아니기때문에 그것들은 자기자체에 의해 분석되여야 한다. 첫번째 함수호출 fib(n-1)은 T의 정의에 의해 $T(N-1)$의 시간단위를 요구한다. 이와 같은 론거는 두번째 함수호출이 $T(N-2)$의 시간단위를 요구한다는것을 보여 준다. 이때 요구되는 전체 시간은 $T(N-1)+T(N-2)+2$인데 여기에서 2는 1행에서의 처리와 3행에서의 더하기를 합한것으로 설명된다. 따라서 $N\geqq 2$에 대하여 fib(n)의 실행시간은 다음과 같다.

$$T(N)=T(N-1)+T(N-2)+2$$

fib(n)=fib(n-1)+fib(n-2)이므로 $T(N)\geqq$ fib(n)을 귀납법으로 증명하는것은 쉽다. 제1장 제2절 5에서 $fib(N)<(5/3)^N$을 증명하였다. 이와 류사하게 $N>4$에 대하여 $fib(N)\geqq(3/2)^N$이라는것을 증명할수 있으며 따라서 이 프로그람의 실행시간은 지수적으로 증가한다는것을 알수 있다. 이 알고리듬은 그리 좋지 못하다. 어떤 간단한 배렬을 만들고 for순환을 리용하면 알고리듬의 실행시간을 충분히 감소시킬수 있다.

이 프로그람은 제1장 제3절에서 서술된 재귀에 대한 4번째 중요규칙(복리규칙)을 위반하는것으로서 거기에서는 방대한 량의 불필요한 처리가 실행되기때문에 속도가 떠지게 된다. 3행에서 첫번째 호출 fib(n-1)은 실제로 어떤 시점에서 fib(n-2)를 계산한다는것을 주의하자. 이 정보는 필요없으며 3행에 있는 두번째 호출에 의해서 다시 계산된다. 무시된 정보의 량은 재귀적으로 증가되며 무한한 실행시간으로 결과가 나타난다. 이것은 물론 《어떤것을 한번이상 계산하지 말라》는 공리의 가장 좋은 실례이지만 사용자는 재귀를 리용하는데 그다지 놀랄것은 없다. 이 책 전반에 걸쳐 재귀에 대한 우수한 리용을 보게 될것이다.

3. 최대부분순서합문제의 풀기

이미전에 취급한 최대부분순서합문제를 풀기 위한 4개의 알고리듬을 고찰하기로 하자. 첫번째 알고리듬은 단지 있을수 있는 모든 가능성들을 남김없이 찾아내는것인데 그것을 프로그람 2-1에 서술하였다. C++에서 for순환은 첨수들을 1이

아니라 0으로부터 시작하여 배렬들을 취급한다. 또한 이 알고리듬은 실제적인 부분렬들을 계산하지 않는데 이것을 수행하기 위하여 추가적인 코드가 요구된다.

```
            /**
            *  Cubic maximum contiguous subsequence sum algorithm.
            */
            int maxSubSum1( const vector<int> & a )
            {
/* 1*/          int maxSum = 0;

/* 2*/          for( int i = 0; i < a.size( ); i++ )
/* 3*/              for( int j = i; j < a.size( ); j++ )
                    {
/* 4*/                  int thisSum = 0;

/* 5*/                  for( int k = i; k <= j; k++ )
/* 6*/                      thisSum += a[ k ];

/* 7*/                  if( thisSum > maxSum )
/* 8*/                      maxSum = thisSum;
                    }

/* 9*/          return maxSum;
            }
```

프로그람 2-1. 알고리듬 1

사용자자신이 이 알고리듬을 수행한다고 생각해 보자(이것은 그리 깊이 생각하지 않아도 될것이다). 그 실행시간은 $O(N^3)$이며 이것은 세개의 계층적인 for순환내부에 들어 있는 $O(1)$의 지령으로 이루어 진 5행과 6행에 의해서 완전히 처리된다. 2행에서의 순환은 N크기를 가진다.

두번째 순환은 크기가 N-i인데 그 크기는 N이거나 그보다 작다. 최악의 경우를 가상하면 마지막한계가 조금 높아 질수 있다는것을 알수 있다. 세번째 순환은 크기 $j-i+1$을 가지는데 이것도 N으로 가상하여야 한다. 전체 분석은 $O(1\cdot N\cdot N\cdot N)=O(N^3)$이다. 1행은 총체적으로 $O(1)$만을 가지고 7행과 8행의 명령도 2중순환의 내부에서는 간단히 처리되기때문에 총체적으로 $O(N^2)$만을 가진다.

이 순환고리들의 실제적크기를 계산하기 위하여 더 정확히 분석해 보면 그 결과는 $\Theta(N^3)$이며 우와 같은 평가는 결수가 6으로서 너무 높다는것을 알수 있다 (상수는 문제로 되지 않으므로 이것은 모두 정확하다.). 이러한 종류의 문제들에 대해서는 이것이 대체로 옳다. 정확한 분석은 합 $\sum_{i=0}^{N-1}\sum_{j=i}^{N-1}\sum_{k=i}^{j}1$로 계산되는데 이것은 6행이 몇번 실행되는가를 나타낸다. 그 합은 제1장 제2절 3에서 고찰한 식들을 리용하여 구체적으로 평가될수 있다. 특히 첫 N개의 옹근수들과 첫 N^2들의 합에 대한 식들을 리용할수 있다. 처음에는

$$\sum_{k=i}^{j}1 = j-i+1$$

이며 다음에는

$$\sum_{j=i}^{N-1}(j-i+1) = \frac{(N-i+1)(N-i)}{2}$$

로 계산한다.

이 합은 바로 첫 N-i개의 옹근수들의 합이라는 사실에 기초하여 계산된다. 계산을 완성하기 위해서

$$\sum_{i=0}^{N-1}\frac{(N-i+1)(N-i)}{2} = \sum_{i=1}^{N}\frac{(N-i+1)(N-i+2)}{2}$$

$$= \frac{1}{2}\sum_{i=1}^{N}i^2 - (N+\frac{3}{2})\sum_{i=1}^{N}i + \frac{1}{2}(N^2+3N+2)\sum_{i=1}^{N}1$$

$$= \frac{1}{2}\frac{N(N+1)(2N+1)}{6} - (N+\frac{3}{2})\frac{N(N+1)}{2} + \frac{N^2+3N+2}{2}N$$

$$= \frac{N^3+3N^2+2N}{6}$$

을 계산한다.

하나의 for순환을 없애여 3차원적인 실행시간을 피할수 있다. 이것은 언제나 가능한것은 아니지만 이 경우에 알고리듬에는 한조의 불필요한 연산표현들이 있다.

개선된 알고리듬에서 수정된 비효률적인 부분은 $\sum_{k=i}^{j} A_k = A_j + \sum_{k=i}^{j-1} A_k$ 를 주의해 보면 알수 있는데 즉 알고리듬 1의 5행과 6행에서의 연산은 지나치게 많은 시간이 든다. 프로그람 2-2는 개선된 알고리듬을 보여 준다. 알고리듬 2는 정확히 $O(N^2)$인데 그 분석은 앞의것보다 더 간단하다.

```
            /**
             * Quadratic maximum contiguous subsequence sum algorithm.
             */
            {
/* 1*/          int maxSum = 0;
/* 2*/          for( int i = 0; i < a.size( ); i++ )
                {
/* 3*/              int thisSum = 0;
/* 4*/              for( int j = i; j < a.size( ); j++ )
                    {
/* 5*/                  thisSum += a[ j ];
/* 6*/                  if( thisSum > maxSum )
/* 7*/                      maxSum = thisSum:
                    }
                }
* 8*/           return maxSum;
            }
```

프로그람 2-2. 알고리듬 2

이 문제를 푸는데는 상대적으로 복잡하고 재귀적인 $O(N\log N)$풀이법도 있는데 그것을 고찰해 보자. 만일 선형적인 $O(N)$풀이법이 없다면 이것은 재귀효과에 대한 아주 좋은 실례로 된다. 그 알고리듬은《**분할통치**》방법을 리용한다. 이 방법은 문제를 2개의 등가적인 보조문제로 나누는것인데 이 보조문제들 역시 재귀적으로 해결할수 있다. 이것이 《분할》부분이다. 《통치》단계는 두개의 보조문제들에서의 풀이를 함께 포함하는것으로 이루어 지며 가능한한 적은 량의 추가처리를 진행하여 전체 문제에 대한 풀이에 도달한다.

이 경우에 최대부분순서합은 세개 부분가운데서 어느 하나에 있을수 있다. 즉 입력자료의 왼쪽 절반전체 또는 오른쪽 절반전체에서 발생하거나 왼쪽의 뒤쪽 절반과 오른쪽의 앞쪽 절반으로 구성되는 부분에서 일어 날수 있다. 첫 2개의 경우

는 재귀적으로 풀수 있다. 마지막경우는 첫번째의 뒤쪽 절반에서의 가장 큰 값과 두번째의 앞쪽 절반에서의 가장 큰 값을 찾는 방법으로 얻을수 있다. 이때 이 두개의 합들은 서로 더해 질수 있다. 실례로 다음의 입력을 고찰해 보자.

첫번째 절반	**두번째 절반**
4 -3 5 -2	-1 2 6 -2

첫번째 절반에 대한 최대부분순서합은 6(요소 A_1~A_3)이고 두번째 절반에 대한 최대부분순서합은 8(요소 A_6~A_7)이다.

첫번째 절반에서 마지막요소를 포함하는 최대합은 4(요소 A_1~A_4)이고 두번째 절반에서 첫 요소를 포함하는 최대합은 7(요소 A_5~A_7)이다. 따라서 중간을 거치는 두 절반의 전체 최대합은 4+7=11(요소 A_1~A_7)이다.

이때 우의 실례에서 최대부분순서합을 얻기 위한 3가지 방법들중 가장 좋은 방법은 두개의 절반들로부터 요소들을 포함하는것이라는것을 알수 있다. 따라서 그 결과는 11이다. 프로그람 2-3은 이 방법의 실현을 보여 준다.

```
        /**
        * Recursive maximum contiguous subsequence sum algorithm.
        * Finds maximum sum in subarray spanning a[1eft..right].
        * Does not attempt to maintain actual best sequence.
        */
        int maxSumRec( const vector<int> & a, int left, int right )
        {
/* 1*/      if( left == right )     // Base case
/* 2*/          if( a[ left ] > 0 )
/* 3*/              return a[ left ];
                else
/* 4*/              return 0;
/* 5*/      int center = ( left + right ) / 2;
/* 6*/      int maxLeftSum = maxSumRec( a, left, center );
/* 7*/      int maxRightSum = maxSumRec( a, center +1, right );
/* 8*/      int maxLeftBorderSum = 0, leftBorderSum = 0;
/* 9*/      for( int i = center; i >= left; i-- )
            {
/* 10*/         leftBorderSum += a[ i ];
```

```
/* 11*/                if( leftBorderSum > maxLeftBorderSum )
/* 12*/                    maxLeftBorderSum = leftBorderSum;
                   }
/* 13*/            int maxRightBorderSum = 0, rightBorderSum = 0;
/* 14*/            for( int j = center + 1; j <= right; j++ )
                   {
/* 15*/                rightBorderSum += a[ j ];
/* 16*/              if( rightBorderSum > maxRightBorderSum )
/* 17*/                  maxRightBorderSum = rightBorderSum;
                     }
/* 18*/            return max3( maxLeftSum, maxRightSum,
                                    maxLeftBorderSum + maxRightBorderSum );
        }
        /**
        * Driver for divide-and-conquer maximum contiguous
        * subsequence sum algorithm.
        */
        int maxSubSum3( const vector<int> & a )
        {
            return maxSumRec( a, 0, a.size( ) - 1 );
        }
```

프로그램 2-3. 알고리듬 3

알고리듬 3에 대한 코드에 대해서는 약간한 주해를 주는것이 좋다. 재귀함수에 대한 일반적인 호출형태는 왼쪽과 오른쪽 경계들과 함께 입력배렬을 넘기는것인데 우에서 계산되는 배렬의 부분은 한계가 없다. 한개의 행으로 이루어 진 구동프로그람은 이것을 배렬과 함께 0과 N-1의 한계들을 넘기는 방법으로 설정한다.

1행~4행은 기초조건을 조정한다. 만일 left==right이면 거기에는 한개의 요소가 있게 되는데 그 요소가 부수가 아니면 그자체가 최대부분순서합으로 된다. left>right인 경우에는 N이 부수가 아니면 문제가 성립하지 않는다(코드에서 약간한 혼란이 발생한다고 하여도). 6행과 7행은 두개의 재귀호출을 수행한다. 재귀호출들은 코드에서 이러한 속성을 파괴할수는 있지만 항상 원래보다 문제를 더 작아져 가게 한다는것을 알수 있다. 8~12행들과 13~17행들은 중앙경계선에 린접하는 두개 부분의 최대합들을 계산한다. 이 두개 값들의 합은 두개의 절반부분들의 최

대합이다. 루틴 max3(이것은 보여 주지 않음)은 세가지 가능성들가운데서 가장 큰 것을 되돌린다.

알고리듬 3은 명백히 앞에서의 두개의 알고리듬들보다도 코드화하는데 더 많은 노력을 요구한다. 그러나 더 짧은 코드가 언제나 더 좋은 코드라는것을 의미하지는 않는다. 이미전에 알고리듬의 실행시간을 보여 주는 표에서 고찰한것처럼 이 알고리듬은 입력크기가 가장 작다는것을 제외하고는 모든 경우에 다른 두개의 알고리듬보다 상당히 빠르다.

그 실행시간은 대체로 피보나치수들을 계산하는 프로그람에서와 같은 방법으로 분석된다. $T(N)$이 크기 N을 가지는 최대부분순서합문제를 푸는데 걸리는 시간이라고 하자. N=1이면 프로그람은 1~4행들을 실행하는데 어떤 상수적인 시간량을 가지는데 이것을 하나의 단위로 고찰한다. 따라서 $T(1)=1$이다. 한편 이 프로그람은 두개의 재귀적인 호출과 9행과 17행사이에 있는 두개의 for순환들, 5행과 18행에서와 같이 몇가지 부차적인 작은 량의 처리를 실행하여야 한다. 두개의 for순환들은 보조배렬에 있는 매개 요소에 접근하기 위해 결합되고 그 순환들의 내부에는 상수적인 처리가 있으므로 9~17행에서 소비되는 시간은 $O(N)$이다. 1~5, 8, 13, 18행들에 있는 코드는 모두 상수적인 처리량이며 따라서 $O(N)$에 비하여 무시될수 있다. 나머지 처리는 6행과 7행에서 실행된다. 이 행들은 $N/2$크기를 가지는 두개의 부분순차문제들을 푼다(N을 짝수라고 가정하면). 따라서 이 행들은 매개가 $T(N/2)$시간단위를 가지며 총체적으로는 $2T(N/2)$의 시간단위를 가진다. 따라서 알고리듬에 대한 전체 시간은 $2T(N/2)+O(N)$이다. 이것은 다음의 식을 준다.

$$T(1)=1$$
$$T(N)=2T(N/2)+O(N)$$

계산을 간단히 하기 위하여 우의 식에서 N을 포함하여 $O(N)$항을 재배치할수 있는데 $T(N)$이 어떤 경우에도 큰O로 표현되게 되므로 이것은 결과에 영향을 주지 않게 된다. 이 식을 정확하게 푸는 방법을 제7장에서 본다. 만일 $T(N)= 2T(N/2)+N$이고 $T(1)=1$이면 $T(2)=4=2*2$, $T(4)=12=4*3$, $T(8)=32=8*4$, $T(16)=80=16*5$이다. 명백하게 유도할수 있는 모형은 $N=2^k$이면 $T(N)=N*(k+1)=N\log N+N=O(N\log N)$이라는것이다.

이 분석은 N이 짝수라고 가정하며 따라서 짝수가 아니면 $N/2$은 정의되지 않는다. 분석의 재귀적인 성질에 의하여 실제로 N이 2의 제곱인 때에만 유효하며 만

일 그렇지 않으면 결과적으로 짝수크기가 아닌 어떤 보조문제를 얻게 되므로 갈기식은 무효로 된다. *N*이 2의 제곱이 아니면 어느 정도 더 복잡한 분석이 요구되지만 그렇다고 하여 큰O결과는 달라 지지 않는다.

다음 장들에서는 재귀에 대한 여러가지 재치 있는 응용들을 보게 된다. 여기서는 최대부분순서합을 찾기 위한 4번째 알고리듬을 보여 주었다. 이 알고리듬은 재귀적인 알고리듬보다 더 간단히 실현되며 더 효과적이다. 이것을 프로그램 2-4에 보여 주었다.

```
           /**
            * Linear-time maximum contiguous subsequence sum algorithm.
            */
           int maxSubSum4( const vector<int> & a )
           {
/* 1*/         int maxSum = 0, thisSum = 0;
/* 2*/         for( int j = 0; j < a.size( ); j++ )
               {
/* 3*/             thisSum += a[ j ];
/* 4*/             if( thisSum > maxSum )
/* 5*/                 maxSum = thisSum;
/* 6*/             else if( thisSum < 0 )
/* 7*/                 thisSum = 0;
               }
/* 8*/         return maxSum;
           }
```

프로그램 2-4. 알고리듬 4

시간한계가 왜 정확한가 하는것은 명백하지만 실제로 그 알고리듬을 왜 써야 하는가를 알려면 생각을 좀 해 보아야 한다. 그 론리를 대략적으로 보기 위해서는 알고리듬 1과 2처럼 i가 현재 서렬의 시작을 가리키고 j는 현재 서렬의 끝을 나타낸다는것에 주의하여야 한다. 실제적으로 가장 좋은 부분렬이 어디에 있는가를 알 필요가 없으면 i의 리용은 그 프로그람을 최적화할수 없게 하지만 이 알고리듬설계에서는 i가 필요하다고 보고 알고리듬 2를 개선한다고 하자. 한가지 방법은 만일 a[i]가 부수이면 그것은 최적부분렬의 시작으로 될수 없기때문에 a[i]에서 시작

하는 부분렬을 a[i+1]에서 시작하는것으로 고칠수 있다. 이와 류사하게 어떤 부수부분렬은 최적부분렬의 선두가 될수 없다. 만일 내부순환에서 a[i]로부터 a[j]까지의 부분렬이 부수라는것을 알게 되면 i를 j로 전진시킬수 있다. 극히 중요한 관찰은 i를 i+1로 전진시킬수 있을뿐아니라 j+1까지 전진시킬수 있다는것이다. 이것을 보기 위하여 p가 i+1과 j사이의 어떤 첨수라고 하자. 첨수 p에서 시작하는 어떤 부분렬은 첨수 i에서 시작하여 a[i]부터 a[p-1]까지의 부분렬을 포함하는 부분렬보다 더 크지 않으므로 그 다음의 부분렬은 부수가 아니다(j는 첨수 i에서 시작하는 부분렬이 부수가 되는 첫번째 첨수이다.). 따라서 i를 j+1로 전진시키는것이 가능하다. 즉 이것은 정확한 풀이를 얻는데 지장이 없다.

이 알고리듬은 재치 있는 많은 알고리듬들가운데서 대표적인것인데 그 실행시간은 명백하지만 정확성은 없다. 이 알고리듬들에서는 거의 언제나 형식적인 정확성립증(앞에서 대략적으로 설명한것보다 더 형식적인)이 요구되는데 아직 많은 사람들이 이에 대하여 확신하지 못하고 있다. 또한 이러한 많은 알고리듬들은 오랜 기간에 걸쳐 더욱 재치 있는 프로그람작성을 요구한다. 그러나 이 알고리듬으로 처리하면 그것들은 고속으로 실행되며 작은 량의 입력을 리용하는 비능률적이며 맹목적인 알고리듬(그러나 쉽게 실현되는)과 비교하는 방법으로 코드론리를 충분히 검사할수 있다.

이 알고리듬의 특별한 우점은 자료를 한번만 통과하며 일단 a[i]를 읽고 처리한 다음에는 그것을 기억할 필요가 없다는것이다. 따라서 배렬이 디스크 또는 테프상에 있으면 그것을 순차적으로 읽을수 있으며 배렬의 어떤 부분을 주기억기에 보관할 필요는 없다. 더우기 어떤 시점에서는 그 알고리듬이 이미 읽어 진 자료의 부분렬문제에 대하여 제때에 정확한 결과를 줄수 있다(다른 알고리듬들은 이 속성을 가지지 않는다). 이러한 알고리듬을 직결(on-line)알고리듬이라고 한다. 일정한 기억공간만을 요구하고 선형시간내에 실행되는 직결알고리듬이 바로 우수한 알고리듬이다.

4. 실행시간의 로그

알고리듬들을 분석하는데서 가장 복잡한것은 로그식이다. 이미 몇가지 분할통치알고리듬들이 $O(N\log N)$시간에 실행된다는것을 보았다. 분할통치알고리듬외에도

로그식들의 빈번한 출현은 다음의 일반적인 규칙에 집중된다. 즉 문제를 어떤 분수크기(이것은 보통 1/2이다.)로 가르는데 상수시간($O(1)$)이 걸린다면 알고리듬은 $O(\log N)$으로 된다. 한편 문제를 단순히 어떤 상수량으로 줄이는데(문제를 1보다 더 작게 하는것과 같은) 상수적인 시간이 필요하다면 그 알고리듬은 $O(N)$이다.

특별한 종류의 문제들에서만 $O(\log N)$으로 될수 있다는것은 명백하다. 실례로 입력이 N개의 수들을 가지는 목록이라고 하면 알고리듬은 순전히 입력을 읽어 들이는데 $\Omega(N)$이 걸려야 한다. 따라서 이러한 종류의 문제들에서 대체로 $O(\log N)$의 알고리듬을 가진다고 하면 그것은 보통 입력이 미리 읽어 진것으로 가정한것이다. 로그성질에 대한 세가지 실례를 보여 준다.

2진탐색

첫번째 실례는 일반적으로 2진탐색과 관련된다.

2진탐색

옹근수 X와 이미 기억기에 정렬되여 있는 옹근수 A_0, A_1, …, A_{N-1}이 있을 때 $A_i=X$인 i를 찾고 만일 X가 입력에 없으면 i=-1을 되돌린다.

명백한 해결방법은 왼쪽으로부터 오른쪽으로 순차적으로 표를 조사해 나가는것인데 이 방법은 선형시간에 실행된다. 그러나 이 알고리듬은 그 목록이 정렬되여 있다는 실제적인 우점을 리용하지 않았으므로 가장 좋은것으로 될수 없다. 더 좋은 방법은 X가 배렬의 중간요소인가를 검사하는것이다. 그것이 옳으면 결과는 인차 알수 있다. 만일 X가 중간요소보다 더 작으면 중간요소의 왼쪽에 있는 정렬된 부분렬에서 같은 전략을 적용할수 있다. 이와 마찬가지로 X가 중간요소보다 더 크면 오른쪽 절반에서 찾는다(거기에는 또한 정지조건도 있어야 한다.). 프로그람 2-5는 2진탐색에 대한 코드를 보여 준다(그 결과는 mid이다.). 여느때와 같이 코드는 C++의 문법적인 습관에 따라 배렬들을 첨수 0에서부터 시작한다.

```
/**
 * Performs the standard binary search using two comparisons per level.
 * Returns index where item is found, or -1, if not found.
 */
```

```
          template <c1ass Comparab1e>
          int binarySearch( const vector<Comparab1e> & a,   const Comparable & x )
          {
/* 1*/        int low = 0, high = a.size( ) - 1;
/* 2*/        whi1e( low <= high )
            {
/* 3*/            int mid = ( low + high ) / 2;
/* 4*/            if( a[ mid ] < x )
/* 5*/                low = mid + 1;
/* 6*/            else if( x < a[ mid ] )
/* 7*/                high = mid - 1;
              else
/* 8*/                return mid;        // Found
            }
/* 9*/        return NOT_FOUND;          // NOT_FOUND is defined as –1
          }
```

프로그람 2-5. 2진탐색

명백히 그 처리는 반복할 때마다 순환의 내부에서 $O(1)$의 실행시간을 가지고 수행된다. 따라서 이 분석은 순환에 대한 반복회수를 결정할것을 요구한다. 순환은 high−low=N-1을 가지고 시작하여 high−low≧−1인 때에 끝난다. 매번 순환할 때마다 high−low값은 그것의 선행값으로부터 적어도 절반씩 줄어 들어야 한다. 따라서 순환의 반복회수는 많아서 $\lceil \log(N-1) \rceil$2이다(실례로 만일 high−low=128이면 매번 반복한 다음의 high-low의 최대값들은 64, 32, 16, 8, 4, 2, 1, 0, −1이다.). 따라서 실행시간은 $O(\log N)$이다. 이처럼 실행시간에 대하여 재귀형식으로 서술하였지만 이와 같은 맹목적인 노력은 실제로 무엇을 하고 있으며 왜 그렇게 하는가를 리해한다면 불필요한것이다.

2진탐색은 첫번째 자료구조실현으로 고찰될수 있다. 그것은 find연산을 $O(\log N)$시간에 실행하지만 다른 모든 연산(특히 insert연산과 같은)들은 $O(N)$시간을 요구한다. 자료가 정적(말하자면 삽입과 삭제가 허용되지 않는)인 응용들에서는 이 알고리듬이 대단히 쓸모 있다. 이때 입력은 먼저 정렬되여야 하며 그다음의 접근들은 빨라 지게 된다. 그 하나의 실례는 원소들의 주기표(물리학과 화학에서

볼수 있는)에 대한 정보를 유지하는데 필요한 프로그람이다. 이 표는 새로운 원소들이 드물게 추가되므로 상대적으로 안정하다. 원소이름들은 정렬되여 보관된다. 거기에는 대체로 110개의 원소들만 있기때문에 한개 원소를 찾는데는 많아서 8번의 접근들이 필요하다. 순차탐색을 실행하면 훨씬 더 많은 접근들이 요구된다.

유클리드알고리듬

두번째 실례는 최대공통약수를 계산하는 유클리드의 알고리듬이다. 두개의 옹근수들에 대한 최대공통약수(gcd)는 두수들을 다 나눌수 있는 가장 큰 옹근수이다. 즉 gcd(50, 15)=5이다. 프로그람 2-6의 알고리듬은 $M \geqq N$일 때 gcd(M,N)을 계산한다.(만일 $N>M$ 이면 순환의 첫번째 반복은 그 값들을 치환한다.)

이 알고리듬은 나머지가 0에 도달할 때까지 계산을 련속적으로 수행한다. 마지막의 령이 아닌 나머지가 답이다. 따라서 만일 M=1,989이고 N=1,590이면 나머지들의 서렬은 399, 393, 6, 3, 0이다. 그러므로 gcd(1989, 1590)=3이다. 실례에서 보여 주는것처럼 이것은 빠른 알고리듬이다.

```
            long gcd( long m, long n )
            {
/* 1*/          while( n != 0 )
                {
/* 2*/              long rem = m % n;
/* 3*/              m = n;
/* 4*/              n = rem;
                }
/* 5*/          return m;
            }
```

프로그람 2-6. 유클리드의 알고리듬

앞에서 본것처럼 이 알고리듬의 전체 실행시간을 평가하는것은 나머지들의 서렬이 얼마나 긴가를 결정하는것에 관계된다. 비록 log$N$이 좋은 결과처럼 보이지만 실례에서 나머지가 399로부터 겨우 393에로 갔다는것을 보았기때문에 나머지값이 어떤 상수배로 감소되여야 한다는것은 전혀 명백치 않다. 실제로 그 나머지는 한

번의 반복에서 어떤 상수배로 감소되지 않는다. 그러나 두번 반복한 다음에는 나머지가 초기값의 거의 절반으로 된다는것을 증명할수 있다. 이것은 반복회수가 최대로 $2(\log N)=O(\log N)$로서 그 실행시간을 결정한다는것을 보여준다. 이 증명은 간단하므로 여기서 고찰해 보자. 그것은 다음의 정리로부터 직접 증명된다.

정리 2-1.

만일 $M > N$이면 $M \bmod N < M/2$이다.

증명:

두가지 경우가 있다. 만일 $N \leqq M/2$이면 나머지가 N보다 더 작으므로 이 경우에 정리는 참으로 된다. 다른 경우는 $N>M/2$인 때이다. 그러나 이때 N은 나머지 M-$N<M/2$을 가지고 M으로 다가가므로 정리는 증명된다.

우의 실례에서 $2\log N$이 대략 20정도이고 7번의 연산들만이 실행되였으므로 가능한 가장 좋은 한계로 될지도 모른다. 최악의 경우(이것은 M과 N이 린접한 피보나치수들일 때 이루어 질수 있다.)에는 상수가 대략 $1.44\log N$까지 약간 개선될수 있다. 유클리드의 알고리듬의 평균경우실행은 고도로 세련된 수학적분석을 요구하며 평균반복회수는 대략 $(12\ln 2\ln N)/\pi^2+1.47$이다.

지수

이 절의 마지막실례는 옹근수가 지수(역시 옹근수)적으로 증가하는 경우를 고찰한다. 지수적으로 표시되는 수들은 일반적으로 아주 크므로 그렇게 큰 옹근수들을 보관할수 있는 기계(또는 이것을 모의할수 있는 번역기)를 가지는 경우에만 이에 대한 분석이 진행된다. 실행시간의 측정은 곱하기의 회수를 계수하는것이다.

X^N을 계산하기 위한 명백한 알고리듬은 N−1번의 곱하기를 리용한다. 프로그람 2-7에 있는 재귀적인 알고리듬은 더 좋은것이다. 1~4행은 재귀의 기초조건을 처리한다. 그리고 N이 짝수이면 $X^N=X^{N/2}\cdot X^{N/2}$로 되며 N이 홀수이면 $X^N=X^{(N-1)/2}\cdot X^{(N-1)/2}\cdot X$로 된다.

```
long pow( long x, int n )
{
```

```
/* 1*/      if( n == 0 )
/* 2*/          return 1;
/* 3*/      if( n == 1)
/* 4*/          return x;
/* 5*/      if( isEven( n ) )
/* 6*/          return pow( x * x, n / 2 );
            else
/* 7*/          return pow( x * x,   n / 2 ) * x;
        }
```

프로그람 2-7. 효과적인 지수계산알고리듬

실례로 X^{62}을 계산하기 위하여 알고리듬은 9번의 곱하기만을 포함하는 다음의 계산을 진행한다.

$$X^3=(X^2)X, \quad X^7=(X^3)^2X, \quad X^{15}=(X^7)^2X, \quad X^{31}=(X^{15})^2X, \quad X^{62}=(X^{31})^2$$

문제를 절반으로 줄이는데 많아서 두번의 곱하기(N이 홀수인 때)가 요구되므로 필요되는 곱하기들의 수는 최대로 $2\log N$이다. 다시말하여 재귀형식이 리용될수 있다. 간단히 말해서 맹목적인 접근은 필요 없다.

정확성에 영향을 주지 않고 코드가 얼마나 수정될수 있는가를 아는것은 때때로 흥미 있는 일이다. 프로그람 2-7에서 3~4행은 실제로 불필요한데 그것은 N이 1이면 7행이 정확히 처리되기때문이다. 7행을

```
/* 7*/      return pow( x, n – 1 ) * x;
```

로 수정하면 프로그람의 정확성에 영향을 주지 않게 된다. 실제로 이 프로그람은 곱하기의 서렬이 앞에서와 같으므로 여전히 $O(\log N)$으로 실행된다. 그러나 6행에 대하여 다음의 방안들은 그것들이 비록 정확해 보인다고 해도 모두 나쁜것들이다.

```
/* 6a*/      return pow( pow( x, 2 ), n / 2 );
/* 6b*/      return pow( pow( x, n / 2 ), 2 );
/* 6c*/      return pow( x, n / 2 ) * pow( x, n / 2 );
```

6a와 6b 두 행들은 N이 2일 때 pow에 대한 재귀호출들중의 하나가 두번째 인수로서 2를 가지기때문에 정확하지 않다. 따라서 처리가 더 진행되지 못하고 무한

순환(종당에는 파괴)에 빠진다.

6c행을 리용하면 현재 $N/2$의 크기를 가지는 재귀호출이 하나가 아니라 두개 존재하기때문에 정확성에 영향을 준다. 분석은 그에 대한 실행시간이 $O(\log N)$보다 더 크지 않다는것을 보여 주게 될것이다. 새로운 실행시간을 결정하기 위하여 그것을 련습문제로 남겨 둔다.

5. 분석검사

일단 분석이 수행되면 결과가 정확하고 좋은가를 보는것이 좋다. 이것을 위한 한가지 방법은 프로그람을 작성하여 경험적으로 주어 진 실행시간이 분석에 의해서 얻어 진 실행시간과 일치하는가를 보는것이다. N이 2배로 되면 실행시간은 선형적인 프로그람에서는 2배, 2차원적인 프로그람에서는 4배, 3차원적인 프로그람에서는 8배로 증가된다. 로그적인 시간에 실행되는 프로그람들은 N이 2배로 되면 그 실행시간이 상수배수만큼 커지며 $O(N\log N)$으로 실행되는 프로그람들은 같은 조건에서 실행하는것보다 두배 길어 진다. 이러한 증가들은 낮은 차수항들이 상대적으로 큰 결수를 가지고 있고 N이 충분히 크지 않을 때에는 조사하기 어려울수 있다. 그 한가지 실례는 최대부분순서합문제의 여러가지 실현들에 대한 실행시간에서 N=10으로부터 N=100으로 뛰여 넘는것이다. 또한 경험적인 검증에 기초하여 $O(N\log N)$프로그람들과 선형적인 프로그람들을 구별하는것은 매우 어렵다.

어떤 프로그람이 $O(f(N))$이라는것을 검증하기 위하여 일반적으로 리용하는 다른 하나의 방법은 N의 범위(보통 2의 배수만큼 간격을 둔)에 대하여 $T(N)/f(N)$값을 계산하는것인데 여기서 $T(N)$은 경험적으로 관측된 실행시간이다. 만일 $f(N)$이 실행시간에 대한 정확한 결과이면 그때 계산된 값들은 어떤 정의상수에로 수렴한다. $f(N)$이 과대평가되면 그 값들은 0으로 수렴한다. $f(N)$이 과소평가된것이고 따라서 옳지 못한 결과이면 그 값들은 발산한다.

실례로 프로그람 2-8의 프로그람토막은 우연적으로 선택된 N보다 크지 않은 두개의 서로 다른 정의옹근수들이 서로 소인수로 될 확률을 계산한다(N이 커지면 그 결과는 $6/\pi^2$에 이른다).

```
double probRelPrime( int n )
{
     int rel = 0, tot = 0;
     for( int i =1; i <= n; i++ )
          for( int j = i + 1; j <= n; j++ )
          {
                 tot++;
                 if( gcd( i, j ) == 1 )
                      rel++;
          }
     return (double) rel / tot;
}
```

프로그램 2-8. 두개의 란수들이 서로 소인수가 될 확률을 평가

사용자는 이 프로그람을 즉시에 분석할수 있다. 표 2-3은 실지 콤퓨터에서 이 루틴에 대하여 관측된 실행시간을 보여 준다. 표는 마지막렬이 가장 일치하며 따라서 사용자가 수행한 분석이 거의 정확하다는것을 보여 준다. 로그식들이 느리게 증가하기때문에 $O(N^2)$과 $O(N^2 \log N)$사이의 차이는 그리 크지 않다.

표 2-3. 프로그람 2-8에 있는 루틴에 대한 경험적인 실행시간들

N	CPU시간(T)	T/N^2	T/N^3	$T/(N^2 \log N)$
100	022	.002200	.000022000	.0004777
200	056	.001400	.000007000	.0002642
300	118	.001311	.000004370	.0002299
400	207	.001294	.000003234	.0002159
500	318	.001272	.000002544	.0002047
600	466	.001294	.000002157	.0002024
700	644	.001314	.000001877	.0002006
800	846	.001322	.000001652	.0001977
900	1,086	.001341	.000001490	.0001971
1,000	1,362	.001362	.000001362	.0001972
1,500	3,240	.001440	.000000960	.0001969
2,000	5,949	.001482	.000000740	.0001947
4,000	25,720	.001608	.000000402	.0001938

6. 분석에 대한 음미

때때로 분석은 과대평가된다는것을 경험적으로 알수 있다. 그러한 경우에는 분석을 보다 더 엄밀하게 하든지(보통 어떤 기묘한 관찰에 의해서) 아니면 평균실행시간이 최악의 경우의 실행시간보다 현저히 작고 또한 그 한계에서의 개선이 불가능하든지 하여야 한다. 많은 복잡한 알고리듬들에서 최악의 경우의 한계는 어떤 나쁜 입력에 의하여 얻을수 있는데 실천적으로는 보통 과대평가된다. 그러나 대부분의 이러한 문제들에서 평균경우의 분석은 대단히 복잡하며(아직 해결되지 않은 많은 경우들에서) 최악의 경우의 한계는 비록 아주 비관적이라고 하더라도 가장 좋은 분석결과로 인정되고 있다.

요약

이 장에서는 프로그람들의 복잡성을 분석하는 몇가지 방법을 주었다. 그러나 그것은 완전무결한것이 아니다. 간단한 프로그람들은 보통 간단히 분석되지만 항상 그런것은 아니다. 실례로 이 책의 뒤에서 정렬알고리듬(쉘정렬, 제7장)들과 **분리모임**(*Disjoint set*)을 유지하기 위한 알고리듬(제8장)을 고찰하게 되는데 이것들은 매개가 약 20개 행의 코드를 요구한다. 쉘정렬의 분석은 아직 완성되지 못하였으며 분리모임알고리듬은 몹시 어렵고 여러 페지에 달하는 복잡한 계산을 요구하는 분석을 가진다. 여기서 취급하게 될 대부분의 분석들은 간단하며 순환을 통한 계산을 포함한다.

아직 취급하지 않은 흥미 있는 분석종류는 **아래한계분석**이다. 제7장에서 이에 대한 실례를 보게 되는데 거기에서는 비교만을 리용하는 방법으로 정렬하는 어떤 알고리듬이 최악의 경우에 $\Omega(N\log N)$의 비교를 요구한다는것이 증명된다. 아래한계 증명들은 하나의 알고리듬이 아니라 문제를 풀기 위한 알고리듬묶음에 적용하기때문에 가장 어렵다.

여기서 보여 주는 몇가지 알고리듬들은 실제적으로 응용할수 있다는것을 언급하는것으로 끝낸다. gcd알고리듬과 지수계산알고리듬은 둘다 **암호학**(*Cryptography*)에 리용된다. 특히 어떤 200자리수는 매개 곱하기후에는 대략 200자리 이하의 수가 얻어 지는 큰 제곱수(보통 또 다른 200자리수자)로 제곱된다. 200

자리수자들을 가지고 처리할것을 요구하므로 효과성이 대단히 중요하다. 지수계산에 대한 간단한 알고리듬은 대략 10^{200}번의 곱하기를 요구하지만 소개된 알고리듬은 최악의 경우 단지 1300번정도의 곱하기만 요구한다.

련습문제

2-1. 증가비률에 따라 다음의 함수들을 정렬하시오.

N, $\sqrt{N}$, $N^{1.5}$, N^2, $N\log N$, $N\log\log N$, $N\log^2 N$, $N\log(N^2)$, $2/N$, 2^N, $2^{N/2}$, 37, $N^2\log N$, N^3

어느 함수들이 같은 비률로 증가하는가를 지적하시오.

2-2. $T_1(N)=O(f(N))$이고 $T_2(N)=O(f(N))$이라고 하자. 다음식들가운데서 어느것이 참인가?

ㄱ. $T_1(N) + T_2(N) = O(f(N))$

ㄴ. $T_1(N) - T_2(N) = o(f(N))$

ㄷ. $\dfrac{T_1(N)}{T_2(N)} = O(1)$

ㄹ. $T_1(N) = O(T_2(N))$

2-3. 어느 함수가 더 빨리 증가하는가, $N\log N$인가, $N^{1+\varepsilon/\sqrt{\log N}}$인가($\varepsilon>0$)?

2-4. 어떤 상수 k에 대하여 $\log^k N=o(N)$임을 증명하시오.

2-5. $f(N)=O(g(N))$도 아니고 $g(N)=O(f(N))$도 아닌 두개의 함수 $f(N)$과 $g(N)$을 찾아 내시오.

2-6. 최근에 어느 한 도시에서 재판관이 한 시민을 인격모욕죄로 재판하고 첫날에 2$의 벌금을 물것을 지시하였다. 그 시민이 재판관의 명령을 집행할 때까지 벌금은 매일 전날의 벌금액수의 두제곱으로 늘어 났다.(즉 그 벌금은 2$, 4$, 16$, 256$, 65,536$,…와 같이 늘어 났다.)

ㄱ. N일 지나서 그 벌금액은 얼마인가?

ㄴ. 그 벌금이 D딸라에 도달되는것은 며칠만인가?(큰O결과로 고찰하시오.)

2-7. 다음의 6개의 프로그램토막들에 대하여

ㄱ. 실행시간에 대한 분석을 하시오(큰O로).

ㄴ. 선택하고 싶은 언어로 코드를 서술하고 *N*값을 여러가지로 주면서 그 실행시간을 계산하시오.

ㄷ. 실제적인 실행시간으로 그 분석을 비교하시오.

(1)
```
sum = 0;
for( i = 0; i < n; i++ )
    sum++;
```

(2)
```
sum = 0;
for( i = 0; i < n; i++ )
    for( j = 0; j < n; j++ )
        sum++;
```

(3)
```
sum = 0;
for( i = 0; i < n; i++ )
    for( j = 0; j < n * n; j++ )
        sum++;
```

(4)
```
sum = 0;
for( i = 0; i < n; i++ )
    for( j = 0; j< i; j++ )
        sum++;
```

(5)
```
sum = 0;
for( i = 0; i < n; i++ )
    for( j = 0; j < i * i; j++ )
        for( k = 0; k < j; k++ )
            sum++;
```

```
(6) sum = 0;
    for( i = 1; i < n; i++ )
        for( j = 1; j < i * i; j++ )
            if( j % i == 0 )
                for( k = 0; k < j; k++ )
                    sum++;
```

2-8. 첫 *N*개의 옹근수들에 대한 어떤 우연적인 순렬을 만들어 낸다고 하자. 실례로 {4, 3, 1, 5, 2}와 {3, 1, 4, 2, 5}는 정확한 순렬들이지만 {5, 4, 1, 2, 1}은 어떤 수(1)가 두번 들어 가고 또 다른 수(3)가 없으므로 정확한 순렬이 아니다. 이 루틴은 흔히 알고리듬들을 모의할 때 리용된다. i와 j사이의 옹근수들을 같은 확률로 발생하는 산법 randInt(i,j)를 가진 란수발생기 r가 존재한다고 하자. 여기에 세가지 알고리듬이 있다.

1. a[0]부터 a[N-1]까지 배렬을 다음과 같이 채우시오. 즉 a[i]를 써넣기 위하여 이미전의 a[0], a[1], …, a[i-1]에는 없는 하나의 수를 얻을 때까지 란수를 발생하시오.
2. 알고리듬 (1)과 같은데 used라고 하는 림시배렬을 보유하시오. 어떤 란수 ran이 배렬 a에 처음으로 넣어 지면 used[ran]=true로 설정하시오. 이것은 첫번째 알고리듬에서 a[i]에 란수를 써넣을 때 그 란수가 이미 리용되였는가를 보기 위하여 i개의 가능한 걸음들 대신에 한번의 걸음으로 검사할수 있다는것을 의미한다.
3. a[i]=i+1로서 배렬을 채우시오. 그때

```
for( i = 1; i < n; i++ )
    swap( a[ i ], a[ randInt( 0, i ) ] );
```

이다.

ㄱ. 세개의 알고리듬들이 모두 정확한 순렬만을 발생시키며 모든 순렬들이 류사하다는것을 증명하시오.

ㄴ. 매개 알고리듬의 예상되는 실행시간을 정확히 분석하시오(큰O).

ㄷ. 좋은 평균시간을 얻기 위하여 매 알고리듬을 10번 실행하는 프로

그람을 각각 작성하시오.

프로그람 (1):N=250, 500, 1,000, 2,000

프로그람 (2):N=2,500, 5,000, 10,000, 20,000, 40,000, 80,000

프로그람 (3):N=10,000, 20,000, 40,000, 80,000, 160,000, 320,000, 640,000

ㄹ. 실제적인 실행시간을 가지고 분석을 비교하시오.

ㅁ. 매 알고리듬에서 최악의 경우 실행시간은 얼마인가?

2-9. 너무 길어서 모의할수 없는 실행시간에 대한 평가들로써 표 2-2에 있는 표를 완성하시오. 이 알고리듬들에 대한 실행시간들을 보간하여 백만개의 수들에 대한 최대부분순서합을 계산하는데 요구되는 시간을 평가하시오. 어떤 가정을 해야 하는가?

2-10. 수동계산에 리용하는 대표적인 알고리듬들에 대하여 다음의것을 수행하는데 걸리는 실행시간을 결정하시오.

ㄱ. 두개의 N자리 옹근수들을 더하기

ㄴ. 두개의 N자리 옹근수들을 곱하기

ㄷ. 두개의 N자리 옹근수들을 나누기

2-11. 어떤 알고리듬이 100개의 입력크기에 대하여 0.5ms를 가진다. 만일 실행시간이 다음과 같다면 500개의 입력크기에 대한 시간은 얼마인가?(낮은 차수항들은 무시할수 있다고 가정하시오.)

ㄱ. 선형

ㄴ. $O(N\log N)$

ㄷ. 2차원

ㄹ. 3차원

2-12. 어떤 알고리듬이 100개의 입력크기에 대해 0.5ms를 가진다. 만일 실행시간이 다음과 같다면 1분동안에 처리할수 있는 문제는 어느 정도 큰가?(낮은 차수항들은 무시할수 있다고 가정하시오.)

ㄱ. 선형

ㄴ. $O(N\log N)$

ㄷ. 2차원

ㄹ. 3차원

2-13. $f(x) = \sum_{i=0}^{N} a_i x^i$ 를 계산하는데 시간이 얼마나 요구되는가?

ㄱ. 지수연산을 수행하는 간단한 루틴을 리용하면?

ㄴ. 제2장 제4절 4에 있는 루틴을 리용하면?

2-14. $f(x) = \sum_{i=0}^{N} a_i x^i$ 를 평가하기 위하여 다음의 알고리듬(Horner의 규칙이라고 하는)을 고찰하자

```
poly = 0;
for( i = n; i >= 0: i-- )
    poly = x * poly + a[i];
```

ㄱ. $x=3$, $f(x)=4x^4+8x^3+x+2$에 대하여 이 알고리듬에 의하여 수행되는 매 단계들을 보여 주시오.

ㄴ. 이 알고리듬이 동작하는 리유를 설명하시오.

ㄷ. 이 알고리듬의 실행시간은 얼마인가?

2-15. 옹근수들의 배렬 $A_1<A_2<A_3<\cdots<A_N$에서 $A_i=i$인 어떤 옹근수 i가 존재하는가를 결정하는 효과적인 알고리듬을 작성하시오. 그 알고리듬의 실행시간은 얼마인가?

2-16. 다음의 고찰에 기초하여 또 다른 gcd알고리듬을 작성하시오.($a>b$가 되도록 배치하라.)

ㄱ. 만일 a와 b가 다 짝수이면 $gcd(a, b) = 2gcd(a/2, b/2)$

ㄴ. 만일 a가 짝수이고 b가 홀수이면 $gcd(a, b) = gcd(a/2, b)$

ㄷ. 만일 a가 홀수이고 b가 짝수이면 $gcd(a, b) = gcd(a, b/2)$

ㄹ. 만일 a와 b가 다 홀수이면 $gcd(a, b) = gcd((a+b)/2, (a-b)/2)$

2-17. 다음 문제들에 대한 효과적인 알고리듬들을 서술하시오(실행시간분석들도 함께 진행하시오.).

ㄱ. 최소순서합찾기

*ㄴ. 최소정수부분순서합찾기

*ㄷ. 최대부분순서적찾기

2-18. 수값분석에서 중요한 문제는 어떤 임의의 f에 대하여 방정식 $f(X)=0$의 풀이를 찾는것이다. 만일 함수가 련속이고 $f(low)$와 $f(high)$가 서로 반대

부호를 가지는 그러한 두점 low와 high를 가진다면 뿌리는 low와 high 사이에 존재하여야 하며 그것은 2진탐색으로 찾아 낼수 있다. 파라메터로서 f와 low, high를 가지고 0에 대해서 푸는 함수를 서술하시오. 마감을 결정하기 위해서는 어떻게 하여야 하는가?

2-19. 이 책에 있는 최대련속부분순서합알고리듬들은 실제 수렬에 대한 어떠한 지표도 주지 않는다. 그것들을 수정하여 최대부분순서렬과 실지 수렬에서의 첨수들을 하나의 객체로 되돌리도록 프로그람을 수정하시오.

2-20. ㄱ. 정의옹근수 N이 씨수인가를 결정하는 프로그람을 작성하시오.

ㄴ. N개의 항들에 대하여 최악의 경우 그 프로그람의 실행시간은 얼마인가?(이것은 $O\left(\sqrt{N}\right)$에 수행되여야 한다.)

ㄷ. B가 N에 대한 2진표현에서 비트의 개수라고 하자. B의 값은 얼마인가?

ㄹ. B에 관하여 그 프로그람의 최악의 경우의 실행시간은 얼마인가?

ㅁ. 20bit수와 40bit수가 씨수인가를 결정하는 프로그람의 실행시간들을 비교하시오.

ㅂ. N 또는 B에 관하여 어느것이 보다 합리적인 실행시간을 줄수 있는가? 왜 그런가?

***2-21.** **에라토스테네스의 채**(Sieve of Eratosthenes)는 N보다 작은 모든 씨수들을 계산하는데 리용하는 산법이다. 먼저 2부터 N까지 옹근수들의 표를 만든다. 다음 삭제되지 않은 가장 작은 옹근수 i를 찾고 그것을 출력하며 i, $2i$, $3i, \cdots$를 삭제한다. 이와 같은 처리를 반복해 나가다가 $i > \sqrt{N}$ 이면 알고리듬을 완료한다. 이 알고리듬의 실행시간은 얼마인가?

2-22. X^{62}는 8번의 곱하기만으로 계산될수 있다는것을 증명하시오.

2-23. 재귀를 리용하지 않는 빠른 지수계산루틴을 작성하시오.

2-24. 빠른 지수계산루틴에 의하여 리용되는 곱하기들의 수를 정확히 계산하시오(참고: N에 대한 2진표현을 고찰하시오.).

2-25. 프로그람 A와 B를 분석하여 최악의 경우의 실행시간이 각각 $150N\log_2 N$과 N^2보다 크지 않다는것을 알았다.

ㄱ. 큰값 N(N>10,000)에 대하여 어느 프로그람의 실행시간이 더 좋은 담보를 주는가?

ㄴ. 작은 값 $N(N<100)$에 대하여 어느 프로그람의 실행시간이 더 좋은 담보를 주는가?

ㄷ.. N=1,000에 대하여 어느 프로그람이 평균경우에 더 빨리 실행되는가?

ㄹ. 모든 가능한 입력들에 대하여 프로그람 B는 프로그람 A보다 더 빨리 실행될수 있는가?

2-26. 크기가 N인 배렬 A에서 과반수요소는 $N/2$번이상 출현하는 요소이다.(따라서 배렬에서 **과반수요소**(*Majority element*)는 많아서 하나이다.) 실례로 배렬

3, 3, 4, 2, 4, 4, 2, 4, 4

는 하나의 과반수요소(4)를 가지는데 이와 달리 배렬

3, 3, 4, 2, 4, 4, 2, 4

는 그렇지 않다. 만일 과반수요소가 없으면 프로그람에서 그것을 지적하시오. 여기에 이 문제를 풀기 위한 대략적인 알고리듬이 있다.

먼저 과반수요소후보를 찾는다.(이것은 더 어려운 부분이다.) 이 후보는 과반수요소가 될 가능성이 있는 요소일뿐이다. 다음 걸음에서는 이 후보가 실제로 과반수를 차지하는가를 결정한다. 이것은 바로 배렬에 대한 순차적인 탐색이다. 배렬 A에서 후보를 찾기 위하여 두번째 배렬 B를 만든다. 그다음 A_1과 A_2를 비교한다. 만일 같으면 그중 하나를 B에 추가하고 그렇지 않으면 아무런 처리도 하지 않는다. 그다음 A_3과 A_4를 비교한다. 역시 같으면 그중 하나를 B에 추가하고 그렇지 않으면 아무런 처리도 하지 않는다. 전체 배렬을 읽어 들일 때까지 이 방법을 계속한다. 그다음 B에 대한 후보를 재귀적으로 찾는다. 이것은 A에 대한 후보이다(왜 그런가?).

ㄱ. 재귀를 어떻게 끝내야 하는가?

*ㄴ. N이 홀수인 경우는 어떻게 조종되는가?

*ㄷ. 이 알고리듬들의 실행시간은 얼마인가?

ㄹ. 림시배렬 B를 리용하지 않으려면 어떻게 하여야 하는가?

*ㅁ. 파반수요소를 찾아내는 프로그람을 작성하시오.

2-27. 입력은 기억기에 이미 존재하는 $N*N$행렬의 수들이다. 매개 개별적인 행은 왼쪽에서 오른쪽으로 가면서 증가한다. 매개 개별적인 렬은 우에서 아래로 가면서 증가한다. 수 X가 그 행렬에 있는가를 결정하는 알고리듬을 최악의 경우 $O(N)$이 되도록 작성하시오.

2-28. 정수들의 배렬 a를 가지고 다음의것을 결정하는 효과적인 알고리듬을 작성하시오.

ㄱ. a[j]+a[i]의 최대값 (j ≧ i)

ㄴ. a[j]-a[i]의 최대값 (j ≧ i)

ㄷ. a[j]*a[i]의 최대값 (j ≧ i)

ㄹ. a[j]/a[i]의 최대값 (j ≧ i)

***2-29.** 콤퓨터모형에서 옹근수들이 고정크기를 가진다고 가정하는것이 왜 중요한가?

2-30. 제1장에서 보여 준 단어맞추기문제를 생각해 보자. 가장 긴 단어의 크기가 10개 문자로 고정된다고 가정한다.

ㄱ. R와 C는 단어맞추기에서 행과 렬의 수이고 W는 단어개수라고 할 때 제1장에서 보여 준 알고리듬들의 실행시간은 R, C, W에 관하여 각각 얼마인가?

ㄴ. 단어목록이 이미 정렬되여 있다고 하자. 훨씬 더 좋은 실행시간을 가지는 알고리듬을 얻기 위하여 2진탐색을 리용하는 방법을 설명하시오.

2-31. 2진탐색루틴에서 5행이 low = mid + 1가 아니라 low = mid라고 하자. 그 루틴이 여전히 동작하겠는가?

2-32. 매번의 반복에서 한번에 두가지 비교가 수행되도록 2진탐색을 실현하시오.

2-33. 알고리듬 3(프로그람 2-3)에서 6행과 7행을

```
/*6*/ maxLeftSum = maxSubSum( a, left, center – 1 );
/*7*/ maxRightSum = maxSubSum( a, center, right );
```

로 치환한다고 하자. 루틴이 여전히 동작하겠는가?

***2-34.** 3차원최대부분순서합알고리듬의 내부순환에서 제일 안쪽부분의 코드들

은 $N(N+1)(N+2)/6$번 반복된다. 2차원적인 알고리듬은 $N(N+2)/2$번의 반복을 수행한다. 선형적인 알고리듬은 N번의 반복을 수행한다. 어느 모형이 명백한가? 이 현상에 대한 종합적인 설명을 줄수 있는가?

참고문헌

알고리듬들의 실행시간분석은 knuth에 의해 세가지 부분의 계렬 [5], [6], [7]에서 통속적으로 보여 주었다. gcd알고리듬에 대한 분석은 [6]에 있다. 그 내용에 대하여 초기에 나온 또 하나의 책은 [1]이다.

큰O, 큰Ω, 큰Θ, 작은o에 대한 표현법들은 [8]에서 knuth에 의해 주장되였다. Θ( )의 리용에 대한 일치한 합의는 아직까지 없다. 많은 사람들은 그리 표현적이지 못하다고 해도 $O(\ )$을 쓰는것을 더 좋아한다. 더우기 $O(\ )$는 일부 분석들에서 Ω( )를 쓸 때에도 여전히 아래한계를 나타내는데 리용된다.

최대부분순서합문제는 [3]에 있다. 이 계렬의 책들([2], [3], [4])은 프로그람의 속도를 최적화하는 방법을 보여 준다.

1. A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms,* Addison-Wesley, Reading, Mass., 1974.
2. J. L. Bentley, *Writing Efficient Programs,* Prentice Hall, Englewood Cliffs, NJ., 1982.
3. J. L. Bentley, *Programming Pearls,* Addison-Wesley, Reading, Mass., 1986.
4. J. L. Bentley, *More Programming Pearls,* Addison-Wesley, Reading, Mass., 1988.
5. D. E. Knuth, *The Art of Computer Programming, Vol 1: Fundamental Algorithms,* 3d ed., Addison-Wesley, Reading, Mass., 1997.
6. D. E. Knuth, *The Art of Computer Programming, Vol 2: Seminumerical Algorithms,* 3d ed., Addison-Wesley, Reading, Mass., 1998.
7. D. E. Knuth, *The Art of Computer Programming, Vol 3: Sorting and Searching,* 2d ed., Addison-Wesley, Reading, Mass., 1998.
8. D. E. Knuth, "Big Omicron and Big Omega and Big Theta," *ACM SIGACT News,* 8 (1976), 18-23.

제3장. 목록, 탄창, 대기렬

이 장에서는 가장 간단하고도 기본적인 세가지 자료구조들을 고찰한다. 대체로 모든 중요한 프로그람들에서는 언제나 이 자료구조들가운데서 적어도 어느 하나를 리용하며 탄창은 사용자가 선언하든 선언하지 않든 프로그람에서 언제나 암시적으로 리용된다. 이 장의 가장 중요한 문제들은 다음과 같다.

- 추상자료형(ADT)에 대한 개념
- 목록에 대한 연산들을 효과적으로 실현하기 위한 방법
- 탄창ADT에 대한 소개와 재귀실현에서 그 리용
- 대기렬ADT에 대한 소개와 조작체계 및 알고리듬설계에서 그 리용

이런 자료구조들은 중요하므로 그것들을 실현하는것은 어렵다고 생각할수 있다. 그것들을 코드로 작성하는것은 사실상 쉽다. 가장 어려운것은 적은 수의 행으로 이루어 진 루틴들에 대한 일반용코드를 잘 작성할수 있도록 충분히 숙련하는것이다.

제1절. 추상자료형

추상자료형(ADT:Abstract Data Type)은 어떤 연산들의 모임을 함께 가지는 객체들의 모임이다. 추상자료형은 수학적인 추상의 개념인데 ADT들을 정의할 때 연산들의 모임을 어떻게 실현하는가에 대한 설명은 전혀 없다. 자체의 연산들을 가지는 목록, 모임, 그라프들과 같은 객체들은 옹근수, 실수, 론리값과 같은 자료형들은 물론 추상자료형으로 고찰할수도 있다. 옹근수, 실수, 론리값형들에는 그것들에 대한 처리를 진행하는 연산들이 포함되며 이것은 추상자료형에서도 마찬가지이다. 모임형 ADT에서는 union(두 모임의 합), intersection(두 모임의 적), size(모임의 크기), complement(나머지 모임)연산들과 같은것들이 그러한 연산으로 된다. 또한 두가지 연산 례컨대 union과 find만을 가질수도 있는데 이것은 모임에 대한 다른 ADT를 정의한다.

C++클라스는 상세한 실현을 적당히 은폐하는 ADT들을 실현할수 있다. 따라서 ADT에 대한 어떤 연산을 실행하는데 필요한 프로그람의 임의의 다른 부분은 적당한 함수를 호출하여 그것을 수행할수 있다. 만일 어떤 원인으로 세부적인 실현내용들을 변경하여야 할 필요가 있으면 단순히 ADT연산들을 실현하는 루틴들을 변경시켜 쉽게 할수 있다. 완전한 클라스계에서 이 변경은 프로그람의 다른 부분에 대하여 완전히 공개적이다.

매개 ADT에 대하여 어떤 연산들이 수행되여야 하는가를 규정하는 규칙은 따로 없으며 이것은 설계할 때 결정하게 된다. 오유정정이나 적당한 곳에서의 산법의 중지 역시

프로그람설계자가 결정한다. 이 장에서 고찰하는 세가지 자료구조들은 ADT의 기본적인 실례들이다. 여기에서 매개 자료구조를 어떻게 실현할수 있는가를 여러가지 방법으로 고찰하고 있는데 그것들은 정확히 실현되기만 하면 그 자료구조들을 리용하는 프로그람들에서는 어떤 자료구조가 리용되였는가를 몰라도 된다.

제2절. 목록ADT

여기에서는 A_1, A_2, A_3, ⋯, A_N 형태의 일반목록을 가지고 취급하게 된다. N을 목록의 크기라고 한다. 크기가 0인 특수한 목록을 빈 목록이라고 한다.

빈 목록을 제외하고 모든 목록에서 A_{i+1}은 $A_i(i<N)$의 다음에 있으며(또는 뒤를 따르며) A_{i-1}은 $A_i(i>1)$에 앞선다. 목록에서 첫번째 요소는 A_1이고 마지막요소는 A_N이다. A_1의 앞요소와 A_N의 뒤요소는 정의되지 않는다. 목록에서 요소 A_i의 위치는 i이다. 문제를 간단히 고찰하기 위하여 설명에서는 목록에 있는 요소들을 옹근수들로 가정하지만 일반적으로는 클라스형판에 의해서 쉽게 조종되는 임의의 복잡한 요소들이 쓰일수도 있다.

목록ADT에 대하여 실현하려고 하는 연산들의 모임은 이러한 《정의들》로 결합된것이다. 몇가지 공통적인 연산들로는 문제에 대한 명백한 처리를 수행하는 printList와 makeEmpty연산들, 어떤 항목이 처음으로 발생하는 위치를 되돌리는 find연산, 어떤 요소를 목록의 어떤 위치에 삽입하거나 지적된 위치로부터 삭제하는 insert와 remove연산들, 파라메터로 지적된 어떤 K번째 위치에 있는 요소를 되돌리는 findKth연산들이 있다. 만일 목록이 34, 12, 52, 16, 12로 되여 있으면 find(52)는 3을 되돌리고 insert(x,3)을 주어진 위치의 다음 위치에 삽입하려고 한다면 목록을 34, 12, 52, x, 16, 12로 만들며 remove(52)는 목록을 34, 12, x, 16, 12로 만든다.

물론 특별한 경우에 취급하는것처림 함수에 어울리는 설명은 전적으로 프로그람작성자에게 달려 있다(실례로 우에서 find(1)이 무엇을 되돌리는가 하는것). 또한 next와 previous와 같은 연산들을 추가할수 있는데 그것들은 어떤 위치를 파라메터로 가지고 각각 앞요소와 뒤요소의 위치들을 되돌린다.

1. 배렬에 의한 목록의 간단한 실현

우에서 언급한 모든 지령들은 배렬을 리용하여 정확히 실현할수 있다. 만일 배렬이 동적으로 할당된다고 하여도 목록의 최대크기를 계산하여야 한다. 일반적으로 이것은 높은 파대평가를 요구하는데 그것은 리용할수 있는 기억공간을 랑비한다. 특히 이것은 알려 지지 않은 크기를 가지는 많은 목록들이 있을 때 여러가지 결함을 가지게 된다.

배렬실현은 기대할수록 좋은 선형시간에 처리될수 있는 printList와 find연산들과 상수시간에 처리될수 있는 findKth연산을 할당한다. 그러나 삽입과 삭제는 시간이 걸린다. 실례로 0위치에 삽입(첫번째 위치에 새로운 요소를 넣는것과 같은)하는것은 먼저 빈칸을 만들기 위하여 전체 배렬을 한자리씩 아래로 밀어야 하고 이와 반대로 첫번째 요소를 삭제하는것은 배렬에 있는 모든 요소를 우로 한자리씩 밀어야 하는데 그때문에 이 연산들은 최악의 경우에 $O(N)$시간이 걸린다. 평균경우에 두 연산들에서는 목록의 절반을 이동하여야 하며 따라서 여전히 선형시간이 요구된다. N번의 련속적인 삽입들로 배렬을 간단히 구축할수 있는데 그것은 2차원적인 시간을 요구한다.

삽입과 삭제에 대한 실행시간이 이렇게 느리고 또한 목록의 크기를 미리 알아야 하므로 일반적으로 간단한 배렬들은 목록을 실현하는데 리용되지 않는다. 이 절의 다음 부분에서는 그 대책으로서 련결목록(Linked List)을 고찰한다.

2. 련결목록

삽입과 삭제에 대한 선형적인 값을 피하기 위하여서는 목록이 련속적으로 보관되지 않아도 된다는것을 인식하는것이 필요하며 따라서 만일 그렇지 않으면 목록의 전체 부분들이 이동되여야 할것이다. 그림 3-1은 련결목록에 대한 일반적인 개념을 보여 준다.

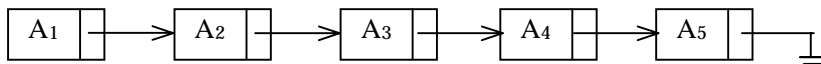



그림 3-1. 하나의 련결목록

련결목록은 여러개의 매듭들로 구성되는데 이것들은 기억기에서 반드시 린접되여 있지는 않다. 매개 매듭은 그의 요소와 뒤요소를 포함하는 어떤 매듭을 가리키는 련결을 포함한다. 이것을 next련결이라고 한다. 마지막기억요소의 next련결은 NULL을 가리킨다.

printList( )나 find( )를 실행하기 위하여 목록의 첫 매듭에서 시작하여 next련결을 따라 그 목록을 순회한다. 이 연산은 명백히 선형시간을 가지는데 배렬실현이 리용되였을 때보다 상수적으로 더 커지게 된다. findKth연산은 배렬실현만큼 더이상 효과적이지 못하다. findKth(i)는 $O(i)$시간을 가지며 또한 명백한 방법을 가지고 목록을 아래로 순회함으로써 수행한다. 실제로 이 한계는 빈번히 i에 의하여 정렬된 순서로 findKth를 호출하기때문에 좋지 못하다. 실례로 findKth(2), findKth(3), findKth(4), findKth(6)은 모두 목록을 아래로 한번씩 순회하여야 실행된다.

remove산법은 하나의 next지적자를 수정하여 실행될수 있는데 그림 3-2는 처음 목록에서 세번째 요소를 삭제한 다음의 결과를 보여 준다.

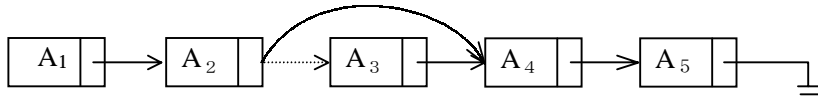



그림 3-2. 련결목록으로부터의 삭제

insert산법은 체계로부터 new연산자를 호출하여 새로운 매듭을 할당하고 다음 두개의 next지적자를 조종하여 실행한다. 그 일반적인 개념을 그림 3-3에 보여 주었다. 여기에서 점선은 낡은 지적자를 나타낸다.

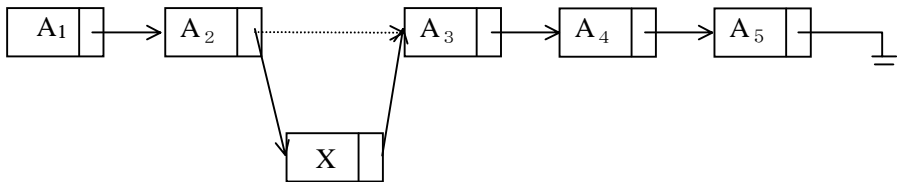



그림 3-3. 련결목록에 삽입

3. 구체적인 프로그람작성

우의 설명은 모든 처리를 진행하는데 실제로 충분하지만 거기에는 주의해야 할 몇가지 문제가 있다. 첫째로, 주어 진 정의로부터 목록의 앞에 어떤 요소를 삽입하는 명백한 방법이 실지 없다. 둘째로, 목록앞에서의 삭제는 하나의 특별한 경우로 되는데 그것은 목록의 시작이 변경되기때문이다. 무관심한 코드작업은 목록을 잃어 버릴수 있다. 세번째 문제는 일반적으로 삭제에 관계된다. 련결을 다음 매듭으로 이동하는것이 간단하다고 하여도 삭제알고리듬은 삭제하려는 매듭의 앞매듭에 대한 자리길을 유지할것을 요구한다.

한가지 간단한 수정을 하여 이 세가지 문제를 모두 해결할수 있다. 그것은 때때로 선두매듭 또는 가상매듭이라고 하는 하나의 감시매듭을 설정하는것이다. 이것은 앞으로 여러번 고찰하게 되는 일반적인 수법이다. 선두매듭은 0위치에 있다고 약속한다. 그림 3-4에 선두매듭을 가진 목록 A_1, A_2,⋯,A_5를 표현한 련결목록을 보여 주었다. 그림 3-5는 빈 련결목록을 보여 준다.

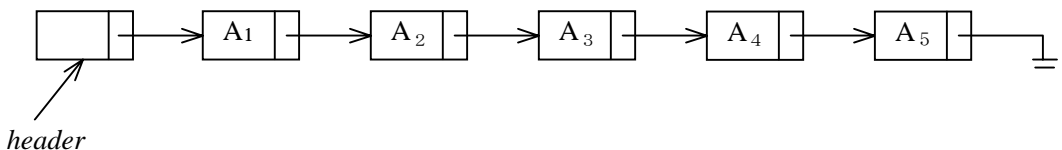



그림 3-4. 선두지적자를 가진 련결목록

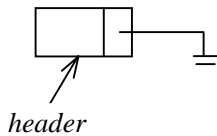



그림 3-5. 선두지적자를 가진 빈 목록

삭제할 때 발생하는 문제들을 피하기 위해서는 findPrevious루틴을 서술하여야 하는데 그것은 삭제하려는 기억요소의 앞요소의 위치를 되돌린다. 선두매듭을 리용하는 경우에는 findPrevious는 목록에서 첫번째 요소를 삭제하려고 할 때 선두매듭의 위치를 되돌린다.

선두매듭의 리용은 어느 정도 론쟁거리로 되고 있다. 일부 사람들은 가상적인 기억요소를 추가하여야 할 충분한 리유가 없다고 주장하는데 그들은 선두매듭의 리용을 원래의것을 다듬는것보다 못하다고 본다. 그렇지만 여기에서는 선두매듭들을 리용하는데 그것은 특별한 경우에 기본적인 련결조작들에 대한 코드를 쉽게 고찰하기 위해서이다. 그밖의 점에서 선두매듭의 리용은 자기자신이 선택하여야 할 문제이다.

실례로 완전한 목록ADT(연산들의 한개 부분모임에 대한)를 고찰하자. 우의 설명에서 이야기된것처럼 목록ADT는 3개 부분의 클라스들로 실현된다. 즉 한개의 클라스는 목록(List) 그자체이고 다른 클라스는 매듭(ListNode)을 나타내며 세번째 클라스는 위치(ListItr)를 표현한다.

프로그람 3-1은 **매듭클라스** ListNode이다. 이 클라스는 두개의 자료성원들 즉 보관된 요소와 다음 매듭에 대한 련결로 이루어 져 있다. 방법들로는 단지 구축자들만 있다. ListNode의 자료성원들이 은페되였다는것에 주의하여야 한다. 그러나 List와 ListItr는 이 자료성원들에 접근하는것이 필요하다. 이를 위하여 ListNode는 List와 ListItr클라스들이 동료들이라는것을 선언한다. 어떤 클라스의 동료는 클라스의 비공개부에 대한 접근을 허용한다. 이것은 이 클라스들이 ListNode내부의 상세한 내용을 볼수 있는 한가지 접근방법으로는 되지만 그것을 대신하지는 않는다. 형판을 실례를 들어 설명하는것이 필요하다. friend선언은 추가적인 문법적습관이 요구된다. 즉 List와 ListItr는 아직 선언되지 않았으므로 번역기는 형판확장들인 List<object>와 ListItr<object>를 혼돈할 수 있다. 이 행들은 클라스형판들이 존재하며 그 구체적인 내용들은 후에 제공된다는것을 의미한다. 그러나 번역기는 friend선언의 의미를 충분히 리해하지는 않는다.

```
template <class Object>
class List;             // Incomplete declaration.
template <class Object>
class ListItr;          // Incomplete declaration.
template <class Object>
class ListNode
{
```

```
        ListNode( const Object & theElement = Object( ), ListNode* n = NULL )
              : element( theElement ), next( n ) { }
        Object       element;
        ListNode *next;
        friend class List<Object>;
        friend class ListItr<Object>;
};
```

프로그람 3-1. 련결목록매듭에 대한 형선언

다음으로 프로그람 3-2에서는 위치에 대한 개념 즉 ListItr를 실현하는 클라스를 보여 준다. 이 클라스를 **반복자클라스**라고 하는데 그것을 간단히 보기로 하자. ListItr는 목록을 끝까지 반복처리하는데 리용할수 있는 산법들을 제공하며 반복자의 현재위치를 나타내기 위하여 ListNode에 대한 참조를 보관한다. isPastEnd는 그 위치가 목록의 끝을 지날 때 참으로 되고 retrive는 현재위치에 보관된 요소를 되돌리며 advance는 현재 위치를 다음 위치에로 전진시킨다. ListItr에 대한 구축자는 현재의 매듭에 대한 지적자를 요구한다. 이 구축자가 비공개부에 있고 따라서 의뢰산법들에 리용될수 없다는것에 주의하여야 한다. 대신에 그 일반적인 개념은 List클라스가 이미 전에 적당히 구축된 ListItr객체들을 되돌린다는것이다. 즉 List는 그 클라스의 동료이며 따라서 ListItr구축자의 비밀을 List에 적용할수 없다. 그러나 그것은 반복자들에 대한 vector를 가지는것이 불가능하며 또한 반복자를 자료성원으로 보관하는 클라스에 대한 복잡한 문제도 제기한다. 따라서 여기에서는 ListItr에 대한 암시적인 구축자를 제공하지만 그 리용은 일반적으로 편리한 문제이다. ListItr클라스의 산법들이 기본적으로 도무 보잘것 없으므로 그것들을 직결방식으로 실현하기 위한 일반적인 단계를 가진다.

```
template <class Object>
 class ListItr
 {
   public:
      ListItr( ) : current( NULL )
      {
      }
      bool isPastEnd( ) .const
      {
            return current == NULL;
      }
      void advance( )
      {
            if( ! isPastEnd( ) )
                current = current->next;
      }
```

```
    const Object & retrieve( ) const
    {
          if( isPastEnd( ) )
              throw Badlterator( );
          return current->element;
    }
  private:
    ListNode<Object> *current;        // Current position

    ListItr( ListNode<Object>    *theNode ) : current( theNode ) { }
    friend class List<Object>;            // Grant access to constructor
};
```

프로그람 3-2. 련결목록에 대한 반복자클라스

프로그람 3-3에서는 **List클라스**의 골격을 보여 주었다. 자료성원은 구축자에 의해서 할당된 선두매듭에 대한 지적자이다. isEmpty는 짧은 하나의 행으로 쉽게 실현된다. zeroth와 first산법들은 각각 선두와 첫번째 요소에 대응하는 반복자들을 되돌린다. 이 루틴들을 프로그람 3-4에 보여 주었다. 다른 루틴들은 목록에서 어떤 항목에 대한 탐색과 삽입 또는 삭제에 의하여 목록을 수정하는것인데 뒤에서 보여 주었다.

```
template <class Object>
class List
{
  public:
    List( );
    List( const List & rhs );
    ~List();

    bool isEmpty ( ) const;
    void makeEmpty( );
    ListItr<Object> zeroth( ) const;
    ListItr<bject> first( ) const;
    void insert( const Object & x, const Listltr<Object> & p )
    ListItr<Object> find( const Object & x ) const;
    ListItr<Object> findPrevious( const Object & x ) const;

    void remove( const Object & x );
    const List & operator=( const List & rhs );
  private:
    ListNode<Object> *header;
};
```

프로그람 3-3. 목록클라스대면부

```
/**
  * Construct the list.
  */
template <class Object>
List<Object>::List( )
{
        header = new ListNode<Object>;
}

/**
  * Test   if the list is logically empty,
  * Return true if empty, false otherwise.
  */
template <class Object>
bool List<Object>::isEmpty( ) const
{
      return header->next == NULL;
}

/**
  * Return an iterator representing the header node,
  */
template <class Object>
ListItr<Object> List<Object>::zeroth( ) const
{
      return ListItr<Object>( header );
}


/**
  * Return an iterator representing the first node   in the list.
  * This operation   is valid for empty lists.
  */
template<class Object>
ListItr<Object> List<Object>::first( ) const
{
         return ListItr<Object>( header->next );
}
```

프로그람 3-4. 목록클라스에 대한 몇개의 효과적인 산법들

프로그람 3-5에 List와 ListItr클라스들이 어떻게 호상작용하는가를 보여 주었다. printList함수는 목록의 내용들을 출력한다. 이 함수는 공개부산법들만 리용하며 시작위치(first에 의한), 끝위치(ispastEnd에 의한)를 지나지 않는가에 대한 검사, 매개 반복적인 조작의 전진(advance에 의한)들을 얻기 위한 전형적인 반복서렬을 리용한다.

```
//Simple print function
 template <class Object>
 void printList( const List<Object> & the List)
 {
       if( theList.isEmptyC ) )
              cout << "Empty list" << endl;
       else
       {
              ListItr<Object> itr = theList.first( );
              for( ; !itr.isPastEnd( ); itr.advance( ) )
                     cout « itr.retrieve( )<< " ";
       }
       cout << endl;
 }
```

프로그람 3-5. 목록을 출력하는 함수

중요한 문제는 세개의 클라스가 모두 실제로 필요한가 필요하지 않는가 하는것이다. 실례로 List클라스는 현재위치에 대한 개념을 정확히 가질수 없는가? 이것은 실행가능한 선택이고 많은 응용들에서 제기되는것이지만 분리된 반복자클라스를 리용하는것은 그 위치와 목록이 실제로는 분리된 객체들이라는 추상성을 나타낸다. 더우기 목록은 여러곳에서 동시에 접근될수 있다. 실례로 어떤 목록에서 하나의 부분목록을 제거하기 위해서는 제거하여야 할 부분목록의 시작과 끝위치들을 지적하기 위하여 두개의 반복자들을 리용하는 목록클라스에 하나의 remove연산을 쉽게 추가할수 있다. 반복자클라스가 없으면 이것은 더 표현하기 어렵다.

이제는 List의 나머지산법들을 실현할수 있다. 먼저 find연산을 보자. 프로그람 3-6에서 보여 준것처럼 그것은 목록에서 어떤 요소의 위치를 돌려 준다. 2행은 론리곱하기(&&)연산이 중단된다는 실제적인 우점을 가지고 있다. 즉 론리곱하기연산의 첫 절반이 거짓이면 그 결과는 자동적으로 거짓으로 되여 두번째 절반은 실행되지 않는다.

어떤 프로그람작성자들은 find루틴을 재귀적으로 코드화하는것을 매력적인것으로 생각하는데 아마 그 리유는 그것이 제멋대로인 마감조건을 피하기때문일것이다. 앞으로 이것은 아주 나쁜 생각이며 어떤 일이 있어도 피해야 한다는것을 고찰하게 된다.

다음 루틴은 목록 *L*로부터 어떤 요소 x를 삭제하는것이다. 여기서는 x가 한번이상 발생하거나 또는 목록에 x가 없을 때 어떤 처리를 하겠는가를 결정하여야 한다. 이 루틴은 처음으로 발생하는 x를 삭제하거나 x가 목록에 없으면 아무러한 처리도 하지 않는다. 이를 위하여 findPrevious를 호출하여 x를 포함하는 매듭의 앞매듭 p를 찾는다. 이것을 실현하는 코드는 프로그람 3-7에 보여 주었다. findPrevious루틴은 find와 류사하며 이것을 프로그람 3-8에 보여 주었다.

```
        /**
          * Return iterator corresponding to the first node containing an item x,
          * Iterator "isPastEnd if item is not found.
          */
        template <c1ass Object>
        ListItr<Object> List<Object>::find( const Object & x ) const
        {
/* 1*/      ListNode<Object> *itr = header->next;
/* 2*/      whi1e( itr != NULL && itr->e1ement != x )
/* 3*/              itr = itr->next;
/* 4*/      return ListItr<Object>( itr );
        }
```

프로그램 3-6. find 루틴

```
/**
  * Remove the first occurrence of an item x.
  */
template <c1ass Object>
void List<Object>::remove( const Object& x )
{
    ListItr<Object> p = findPrevious( x );
    if( p.current->next != NULL )
    {
        ListNode<Object> *o1dNode = p.current->next;
        p.current->next = p.current->next->next; // Bypass deleted node
        delete oldNode;
    }
}
```

프로그램 3-7. 련결목록에서의 삭제루틴

```
        /**
          *   Return iterator prior to the first node containing an item x.
          */
        template<c1ass Object>
        ListItr<Object> List<Object>::findPrevious( const Object & x ) const
        {
/* 1*/      ListNode<Object> *itr = header;
/* 2*/      whi1e( itr->next != NULL && itr->next->e1ement != x )

/* 3*/              itr = itr->next;
/* 4*/      return ListItr <Object>( itr );
        }
```

프로그램 3-8. remove 루틴에 리용하기 위한 find 루틴-findPrevious

서술하여야 할 마지막루틴은 삽입루틴이다. 삽입되여야 할 요소와 어떤 위치 p를 넘긴다. 개별적인 삽입루틴은 p에 의해서 지적된 위치의 다음 위치에 하나의 요소를 삽입하게 된다. 이에 대한 결정은 자의적이며 삽입수행에 대한 규칙들이 따로 정해 진것이 없다. 새로운 요소를 p위치에 삽입(이것은 p위치에 있는 요소의 앞을 의미)하는것은 전적으로 가능하지만 이것을 수행한는것은 p위치의 앞에 있는 요소에 대한 정보를 요구하게 된다. 이것은 findPrivious를 호출하여 수행할수 있다. 따라서 처리하려는 내용을 설명하는것이 중요하다. 이것은 프로그람 3-9에서 처리된다.

```
/**
* Insert item x after p.
*/
template <c1ass Object>
void List<Object>::insert( const Object & x, const ListItr<Object> & p )
{
    if( p.current != NULL )
        p.current->next = new ListNode<Object>( x, p.current->next );
}
```

프로그람 3-9. 련결목록에 대한 삽입루틴

insert루틴은 자기가 속해 있는 목록을 리용하지 않는데 그것은 오직 p에만 관계된다는것을 주의하여야 한다. 반복자가 목록에 대응한다는것을 확인하기 위한 검사는 련습으로 남겨 둔다. 이것은 그 목록에 대한 참조를 목록반복자에 대한 림시 자료성원으로 추가하는 방법으로 수행된다.

find와 findPrevious루틴(그리고 findPrevious를 호출하는 remove루틴)들을 제외하고 모든 연산들은 $O(1)$시간이 걸리도록 코드화된다. 이것은 모든 경우에 고정된 수의 지령들만 실행되므로 목록이 얼마나 큰가 하는것은 문제로 되지 않는다. find와 findPrevious루틴들에 대한 실행시간들은 최악의 경우에 $O(N)$인데 그것은 요소를 찾지 못하거나 목록의 마지막에 있을 때 전체 목록이 순회되여야 하기때문이다. 평균경우에도 실행시간은 $O(N)$으로 되는데 이것은 이 경우에 목록이 절반은 순회되여야 하기때문이다.

4. 기억기재리용과 3대요소

삽입루틴이 언제나 new연산자를 호출하여 ListNode객체들을 할당하기때문에 이 객체들이 더이상 필요하지 않을 때에는 그것을 재리용하는것이 중요하다. 그렇지 않으면 제1장에서 서술된것처럼 기억기루실이 발생할수 있다. 이것은 delete연산자를 호출하여 처리

할수 있다. 이것을 처리하기 위한 여러가지 산법들이 있는데 그것은 한개 매듭을 제거하는 remove산법과 *N*개 매듭들을 제거하는 makeEmpty산법, 선두매듭을 포함하여 *N*+1개의 매듭들을 제거하는 해체자들이다.

프로그람 3-7에서 그에 대한 일반적인 수법 즉 참조할 필요가 없는 매듭에 대한 지적자를 절약하는 방법을 보게 된다. 지적자조작들은 삭제할 매듭을 띄여 넘은 다음 delete를 호출한다. 이 지령은 중요하다. 즉 일단 어떤 매듭이 delete로 처리되면 그 내용들은 불안정하다. 《불안정》하다는것은 그 매듭이 앞으로 new요구를 충족시키는데 리용될수 있다는것을 의미한다. 이것은 문제들을 처리하는데 어떤 번역기를 선택하는가에 관계되기때문에 프로그람 3-7에서 delete지령을 우로 한행 이동하는것은 역효과를 가지지 않지만 그래도 역시 그것은 정확하지 않다는것을 의미한다. 사실 이것은 최악의 오유를 유도한다. 즉 그것은 때때로 부정확한 작용을 줄뿐이다.

*N*개의 매듭들을 제거하는 makeEmpty와 *N*+1개의 매듭들을 제거하는 해체자는 더 복잡하다. 그러나 기억기재리용이 흔히 복잡하므로(그리고 많은 비률의 C++오유들을 발생시킬수 있으므로) 가능한껏 delete리용을 피하는것이 좋다. makeEmpty에서는 이것을 첫번째 요소에 대하여 목록이 빌 때까지 remove를 반복적으로 호출하여 처리할수 있다. 따라서 기억기재리용은 remove에 의해서 자동적으로 조종된다. 해체자에서 makeEmpty를 호출하고 그다음 선두매듭에 대하여 delete를 호출한다. 이 두 루틴들을 프로그람 3-10에서 보여 준다.

```
/**
 * Make the list logically empty.
 */
 template <class Object>
 void List<Object>::makeEmpty( )
 {
        while( !isEmpty( ) )
               remove( first( ).retrieve( ) );
 }
 /**
   * Destructor
   */
 template <class Object>
 List<Object>::~List( )
 {
       makeEmpty( );
              delete header;
 }
```

프로그람 3-10. makeList와 List해체자

제 1장에서 암시적인 해체자를 받아 들일수 없을 때는 복사대입연산자(operator=)와 복사구축자도 받아 들일수 없다는것을 보여 주었다. operator=에 대하여 공개부목록산법들을 가지는 항목들에서 간단한 실현을 줄수 있다. 이것을 프로그람 3-11에 보여 주었다. 그것은 일반적인 경계(aliasing)를 검사하고 *this를 되돌린다. 복사하기에 앞서 이미전에 목록에 할당되였던 기억기의 류실을 피하기 위하여 현재목록을 비게 한다. 빈 목록을 가지고 첫번째 매듭을 만들며 그다음 목적하는 목록의 끝까지 새로운 ListNode들을 추가하는 rhs처리를 계속한다.

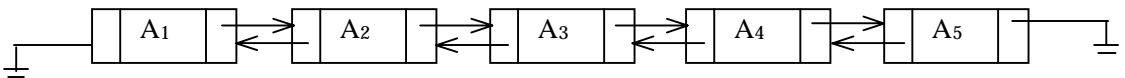
복사구축자에서는 프로그람 3-11에서 보여 준것처럼 선두매듭을 할당하기 위하여 new를 호출하고 그다음 rhs를 복사하기 위하여 operator=를 리용하는 방법으로 하나의 빈 목록을 만들수 있다. 일반적으로 리용되는 수법은 List가 값(상수참조 대신에)에 의한 호출을 리용하여 넘겨 질 때 번역기가 오유통보문을 발생시킨다는 개념을 가지고 복사구축자를 비공개부로 만든다.

```
/**
 * Deep copy of linked lists.
 */
template <c1ass Object>
const List<Object> & List<Object>::operator=( const List<Object> & rhs )
{
    if( this != &rhs )
    {
        makeEmpty( );
        ListItr<Object> ritr = rhs.first( );
        ListItr<Object> itr = zeroth( );
        for( ; !ritr.isPastEnd( ); ritr.advance( ), itr.advance( ) )
            insert( ritr.retrieve( ), itr );
    }
    return *this;
}
/**
 * Copy constructor.
 */
template <c1ass Object>
List<Object>::List( const List<Object> & rhs )
{
    header = new ListNode<Object>;
    *this = rhs;
}
```

프로그람 3-11. List복사루틴들: operator=와 복사구축자

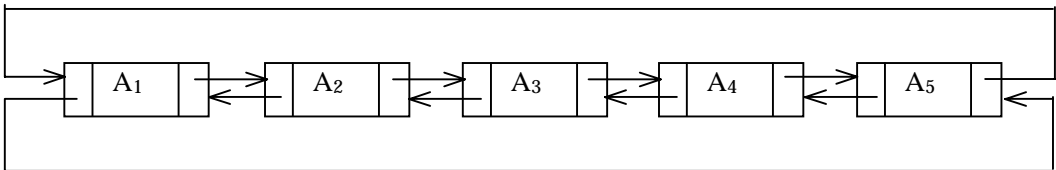
5. 2중련결목록

때때로 목록을 거꾸로 순회하는것이 편리하다. 표준적인 실현은 이것을 지원하지 않지만 그 해결은 간단하다. 그 방법은 단순히 앞매듭에 대한 련결을 보관하는 또 하나의 자료성원을 매듭에 추가하는것이다. 이 특별한 련결값은 기억공간을 요구하며 련결들을 더 많이 설치하여야 하므로 삽입과 삭제들의 비용이 배로 늘어 난다. 한편 그것은 앞매듭에 대한 지적자를 리용함으로써 어떤 항목에 대하여 더이상 참조하지 않아도 되므로 삭제가 간단하게 진행되는데 이 정보는 현재 준비되여 있다. 그림 3-6은 2중련결목록을 보여 준다.

그림 3-6. 2 중련결목록

6. 순환련결목록

일반적으로 마지막매듭에서 다음 매듭에 대한 뒤련결이 첫번째 매듭을 가리키도록 약속한다. 이것은 선두매듭이 있든 없든 처리할수 있으며(만일 선두매듭이 있으면 마지막매듭은 그것을 련결한다.) 또한 2중련결목록을 가지고 할수도 있다(첫번째 매듭의 앞련결은 마지막매듭을 가리킨다.). 이것은 명백히 일부 검사들에 어떤 영향을 주지만 이 자료구조는 여러 응용들에서 일반적이다. 그림 3-7은 선두매듭이 없는 **2중순환련결목록**를 보여 준다.

그림 3-7. 2중순환련결목록

7. 실례

여기서는 련결목록들을 리용한 세가지 실례를 보여 준다. 첫번째 실례는 한변수다항식들을 표현하는 간단한 방법이다. 두번째 실례는 어떤 특별한 경우 선형시간에 자료들을 정렬하는 산법이다. 마지막으로 종합대학에서 과정등록자들을 기록하기 위하여 어

떤 련결목록을 리용하겠는가 하는 복잡한 실례이다.

다항식ADT

목록을 리용하여 부아닌 지수들을 가지는 한변수다항식들에 대한 추상자료형을 정의할수 있다. $f(x)=\sum_{i=0}^{N} a_i x^i$ 라고 하자. 대부분의 결수 a_i들이 령이 아니라면 결수들을 보관하기 위하여 간단한 배렬을 리용할수 있다.

그다음 이 다항식들에 대한 더하기, 덜기, 곱하기, 나누기와 기타 다른 연산들을 실현하기 위한 루틴들을 서술한다. 이 경우에 프로그람 3-12에 주어 진 형선언들을 리용하여야 한다. 그다음 변수연산들을 실현하기 위한 루틴들을 서술해야 한다. 두가지 가능성은 더하기와 곱하기인데 이것들을 프로그람 3-13부터 프로그람 3-15까지에서 보여 준다. 출력다항식들을 령으로 초기화하는 시간을 무시하면 곱하기루틴의 실행시간은 두 입력다항식들의 차수에 비례한다. 이것은 많은 항목으로 이루어 진 조밀한 다항식들에서는 그렇게 되지만 만일 두 다항식 $p_1(x)=10x^{1000}+5x^{14}+1$과 $p_2(x)=3x^{1990}-2x^{1492}+11x+5$이 입력되면 그 실행시간은 접수할수 없다. 그것은 령으로의 곱하기들과 입력다항식들의 실제하지 않는 부분들에 대한 계산에 많은 시간이 소비되기때문에 언제나 비효률적이다.

```
class Polynomial
{
   public:
        Po1ynomia1( );
        void insertTerm( int coef, int exp );
        void zeroPolynomial( );
        Polynomial operator+( const Polynomial & rhs ) const;
        Polynomial operator*( const Polynomial & rhs ) const;
        void print( ostream & out ) const;
   private:
        static const int MAX_DEGREE = 100;

        vector<int> coeffArray;
        int highPower;
 };
```

프로그람 3-12. 다항식ADT의 배렬실현에 대한 클라스선언

```
Void Polynomial::zeroPolynomial( )
  {
        for( int i = 0; i <= MAX_DEGREE; i++)
                coeffArray[ i ] = 0;
        highPower = 0;
  }
```

프로그람 3-13. 다항식을 0으로 초기화하는 산법

```
Polynomial Polynomial::operator+( const Polynomial & rhs ) const
{
    Polynomial sum;
    sum.highPower = max( highPower, rhs.highPower );
    for( int i = sum.highPower; i >= 0; I-- )
        sum.coeffArray[ i ] = coeffArray[ i ] + rhs.coeffArray[ i ];
    return sum;
}
```

프로그람 3-14. 두개의 다항식을 더하는 산법

```
Polynomial Polynomial:: operator*( const Polynomial & rhs ) const
{
    Polynomial product;
     product. highPower = highPower + rhs.highPower;
     if( product.highPower > MAX_DEGREE )
         throw Overf1ow( );
     for( int i = 0; i <= highPower; i++ )
        for( int j = 0; j <= rhs.highPower; j++ )
             product.coeffArray[ i + j ] += coeffArray[ i ] * rhs.coeffArray[ j ];
     return product;
}
```

프로그람 3-15. 두개의 다항식을 곱하는 산법

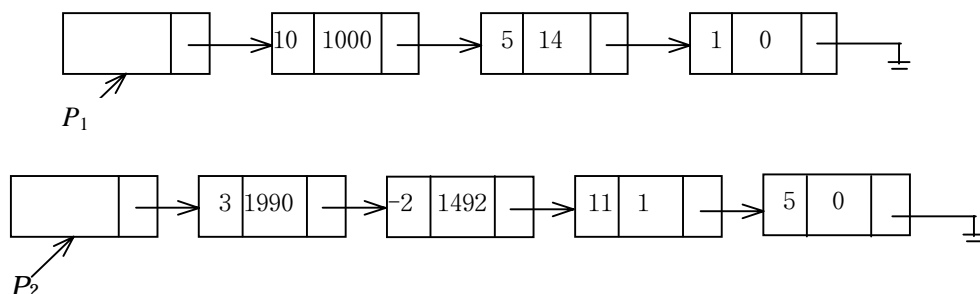



그림 3-8. 두 다항식들을 표현하는 련결목록

방안은 하나의 련결목록을 리용하는것이다. 다항식에 있는 매개 항은 하나의 매듭에 포함되며 매듭들은 지수들이 감소되는 순서로 보관된다. 실례로 그림 3-8에 있는 련결목록은 $P_1(x)$와 $P_2(x)$이다. 그다음 프로그람 3-16에서의 선언을 리용한다.

그다음 연산들을 실현하는것은 간단하다. 다만 잠재적으로 어려운것은 두 다항식들이 곱해질 때인데 다항식의 결과는 결합된 항들과 같아야 한다. 여러가지 방법을 리용하

여 이것을 처리할수 있지만 련습문제로 남겨 둔다.

```
class Literal
{
  private:
      // Various constructors
    int coefficient;
    int exponent;
    Friend class Polynomial;
};
class Polynomial
{
    public:
            Polynomial ( );
            void insertTerm( int coef, int exp );
            void zeroPolynomial( );
            Polynomial operator+( const Polynomial & rhs ) const;
            Polynomial operator*( const Polynomial & rhs ) const;
            void print( ostream & out ) const;
  private:
      List<Litera1> terms;
};
```

프로그람 3-16. 다항식ADT의 련결목록실현에 대한 클라스대면부

밑수정렬

련결목록을 리용하는 두번째 실례는 밑수정렬이다. 밑수정렬은 때때로 카드정렬이라고도 하는데 그것은 이 정렬이 현재콤퓨터의 출현에 이르기까지 오래된 방법의 하나인 착공카드들의 정렬에 리용되였기때문이다.

만일 1부터 M(또는 0~M−1)까지의 범위에 있는 N개의 옹근수들이 있다고 할 때 이 정보를 리용하여 바께쯔정렬이라고 하는 빠른 정렬방법을 얻을수 있다. 0으로 초기화된 count라고 하는 M크기의 하나의 배렬을 유지한다. count는 M개의 요소(또는 바께쯔)들을 가지는데 그것은 초기에 비여 있다. A_i가 읽어 지면 count[A_i]는 1만큼 증가한다. 입력이 모두 읽어 진 다음 정렬된 목록의 상태를 출력하기 위하여 count배렬을 주사한다. 이 알고리듬은 $O(M+N)$을 가지는데 그에 대한 증명은 련습으로 남겨 둔다. 만일 $M=\Theta(N)$이면 그때 바께쯔정렬은 $O(N)$으로 된다.

밑수정렬은 이 방법에 대한 일반화이다. 그 내용을 고찰하기 위한 가장 쉬운 방법은 실례를 들면서 설명하는것이다. 0~999까지의 범위에 있는 10개의 옹근수들을 정렬하려고 한다고 하자. 일반적으로 이 N개의 수들은 어떤 상수 P에 대하여 0부터 N^P−1의 범위에 있다. 이때에는 명백히 바께쯔정렬을 리용할수 없는데 그것은 너무 많은 바께쯔

들이 있어야 하기때문이다. 그 비결은 바께쯔정렬의 여러 단계를 리용하는것이다. 자연적인 알고리듬은 제일 높은 《자리》(습관적으로 N에 기초하는 자리)에 의해 바께쯔정렬을 진행하고 그다음 두번째로 높은 자리에 대하여 우와 같은 처리를 진행하며 이렇게 계속 반복하여 수행하는것이다. 이 알고리듬은 매개 바께쯔를 재귀적으로 바께쯔정렬하는것이 바께쯔경계들에 대한 자리길을 너무 많이 유지할것을 요구하므로 동작하지 않는다. 그러나 만일 제일 낮은 《자리》에 대해서 먼저 바께쯔정렬들을 실행한다면 그때 알고리듬은 동작한다. 물론 초기의 바께쯔정렬과는 달리 같은 바께쯔에 한개이상의 수가 넣어 지게 되는데 이 수들은 각이하므로 그것들을 하나의 목록에 유지한다. 일반적으로 모든 수들은 여러개의 자리를 가진다. 만일 간단한 배렬이 목록으로 리용되였으면 매개 배렬은 전체 공간요구 $\Theta(N^2)$에 대하여 크기 N을 가지게 된다.

다음의 실례는 10개의 옹근수들에 대한 밑수정렬의 처리과정을 보여 준다. 입력은 우연적인 첫 10개 수자들의 3제곱수들인 64, 8, 216, 512, 27, 729, 0, 1, 343, 125이다. 첫번째 단계는 제일 낮은 자리들에 대한 바께쯔정렬이다. 이 경우에 문제를 간단히 고찰하기 위하여 수학적인 처리는 밑수를 10으로 하지만 이것은 일반적이라고 할수 없다. 표 3-1에서 보여 주는 바께쯔들은 제일 낮은 자리에 대하여 정렬된 목록인데 0, 1, 512, 343, 64, 125, 216, 27, 8, 729이다. 이것들은 이제 다음으로 낮은 자리에 대하여 정렬된다(여기서는 10의 자리이다). 표 3-2를 보시오. 두번째 단계는 0, 1, 8, 512, 216, 125, 27, 729, 343, 64를 출력한다. 이 목록은 이제 두번째로 낮은 자리들에 대해서 정렬된다. 표 3-3에서 보여 주는 마지막단계는 제일 높은 자리에 대한 바께쯔정렬이다. 마지막목록은 0, 1, 8, 27, 64, 125, 216, 343, 512, 729이다.

표 3-1. 밑수정렬의 첫 단계 처리후의 바께쯔

0	1	512	343	64	125	216	27	8	729
0	1	2	3	4	5	6	7	8	9

표 3-2. 밑수정렬에서 두번째 단계처리후의 바께쯔

8		729							
1	216	27							
0	512	125		343		64			
0	1	2	3	4	5	6	7	8	9

이 알고리듬의 동작을 고찰할 때 만일 두개의 수들이 같은 바께쯔에 잘못된 순서로 넣어 지면 정렬이 실패한다는것을 주의하여야 한다. 그러나 앞선 단계들은 어떤 바께쯔에 여러개의 수들이 입력될 때 정렬된 순서로 입력되도록 담보한다. 실행시간은 $O(P(N+B))$인데 여기서 P는 단계들의 수이고 N은 정렬하여야 할 요소들의 수이며 B는 바

께쯔들의 수이다. 우의 경우에서는 $B=N$인데 전형적으로 $B \ll N$이고 P가 상수이면 $O(N)$으로 된다.

표 3-3. 밑수정렬에서 마지막단계처리후의 바께쯔

64 27 8 1 0	125	216	343		512		729		
0	1	2	3	4	5	6	7	8	9

실례로 2^{11}의 바께쯔크기에 대하여 3개의 단계로 처리한다면 밑수정렬로 32bit옹근수들을 모두 정렬할수 있다. 이 알고리듬은 이 콤퓨터에서 언제나 $O(N)$으로 되지만 큰 상수가 포함되므로 아직은 제7장에서 보게 되는 알고리듬만큼은 효과적이지 못하다.($\log N$의 결수는 모두 그렇게 크지 않으며 이 알고리듬은 련결목록을 유지하기 위한 부가적인 비용을 가지게 된다는것을 기억하여야 한다.)

다중목록

마지막실례는 련결구조들에 대한 더 복잡한 리용을 보여 준다. 40,000명의 대학생들과 2,500개의 학과를 가진 어떤 종합대학에서는 두가지 형태의 기록들을 가지고 있을수 있다. 첫번째 기록은 매개 학급에 대한 등록자수를 표시하고 두번째 기록은 매 학생별로 그 학생이 등록되여 있는 학급을 표시한다.

이것을 명백히 실현하기 위해서는 2차원배렬을 리용할수 있다. 그러한 배렬은 1억개의 입구점들을 가진다. 평균적으로 학생들은 대체로 3개 학과에 등록되므로 이 입구점들의 120,000개만 즉 대략 0.1%만이 실제로 의의 있는 자료를 가진다.

필요한것은 그 학과의 학생들을 포함하는 매개 학과에 대한 목록이다. 또한 매개 학생에 대하여 그 학생이 등록된 학과를 포함하는 목록도 필요하다. 그림 3-9는 이 실현을 보여 준다.

그림에서 보여 준것처럼 두개 목록들이 하나로 결합되여 있다. 이러한 기억구조를 다중목록이라고 한다. 모든 목록들은 하나의 선두매듭을 리용하였으며 순환적이다. 학과 C3에 있는 모든 학생들을 표시하기 위하여 C3에서 시작하여 오른쪽으로 가면서 그 목록을 순회한다. 첫번째 요소는 S1학생에 속한다. 거기에 이에 대한 구체적인 정보는 없다고 하여도 그것은 선두매듭에 도달할 때까지 학생의 련결목록을 따르는것으로 결정할수 있다. 일단 이것이 수행되면 C3의 목록을 되돌리고(그 학생에 대한 목록을 순회하기전에 그 위치를 과정안목록에 보관하였다.) S3에 속하는것을 결정할수 있는 또 다른 요소

를 찾는다. 또한 S4와 S5도 이 학과에 있다는것을 계속하여 찾을수 있다. 모든 학생에 대하여 이와 류사한 방법으로 그 학생이 등록되여 있는 학과들을 결정할수 있다.

순환목록을 리용하면 공간은 절약하지만 그만큼 시간소비가 있게 된다. 최악의 경우에 첫 학생이 매개 학과에 등록되였다면 그때 매개 입구점은 그 학생에 대한 학과명칭을 모두 결정하기 위하여 목록을 조사해 볼 필요가 있다. 그것은 이 응용에서 상대적으로 매 학생당 적은 수의 학과들이 그리고 매 학과당 적은 학생수가 있을수 있고 이것은 어떻게 될지 알수 없기때문이다. 만일 이것이 어떤 문제를 발생시킬수 있다고 짐작된다면 선두매듭이 아닌 매개 요소들은 그 학생의 바로 뒤에 대한 련결들과 학과의 선두매듭을 가질수 있을것이다. 이것은 두배의 공간을 요구하지만 그 실현에서는 간단하고 속도가 빠르다.

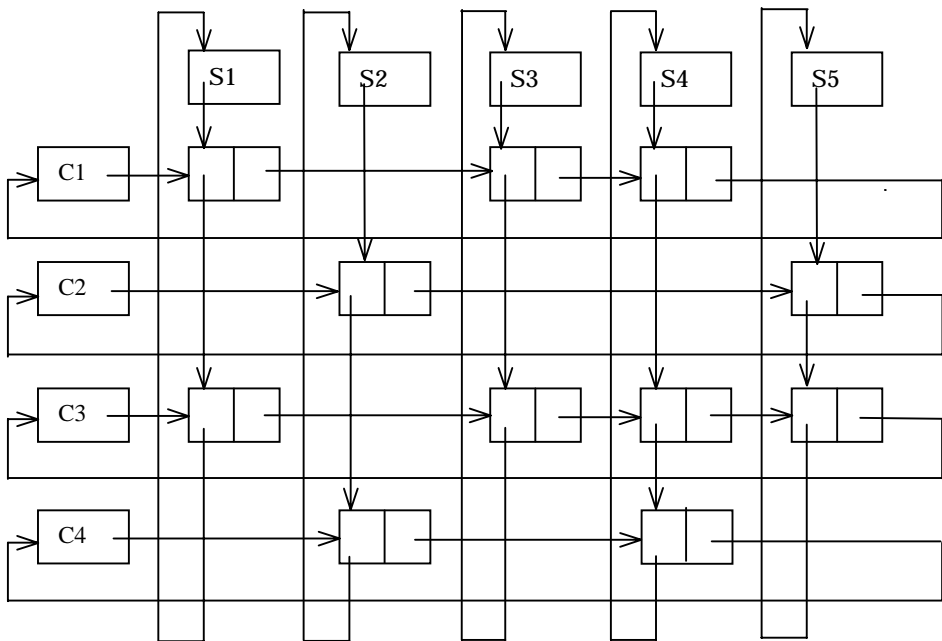



그림 3-9. 등록문제에 대한 다중목록실현

8. 련결목록의 유표적인 실현

BASIC와 FORTRAN과 같은 많은 언어들은 동적련결구조를 제공하지 않는다. C와 C++와 같은 언어들은 new를 반복적으로 호출하는것이 어떤 경우에는 비실용적이라는것을 알수 있다. 때때로 다른 하나의 실현방법이 리용되는데 그것을 유표적인 실현이라고 한다.

련결목록의 실현에서 두가지 중요한 특징은 다음과 같다:

① 자료는 매듭들의 집합으로 보관된다. 매개 매듭은 자료와 다음 매듭에 대한 련결을 포함한다.

② 새로운 매듭은 체계기억기로부터 new를 호출하여 참조할수 있으며 더이상 참조할 필요가 없으면 delete를 호출하여 개선한다.

유표적인 실현은 이것을 모의할수 있어야 한다. 조건 ①을 만족시키기 위한 론리적인 방법은 매듭들에 대하여 static배렬을 가지는것이다. 배렬안의 어떤 요소에서 그 배렬의 첨수는 매듭지적자대신에 리용된다. 프로그람 3-17에서는 련결목록의 유표적인 실현에서의 반복자클라스를 선언한다. 이 코드는 이미전에 고찰한 련결목록클라스에 대한 련결목록의 유표적인 실현을 대응시킨다.

```
template <c1ass Object>
class ListItr
{
   public:
      ListItr( ) : current( 0 )
      {
      }
      bool isPastEnd( ) const
        {
              return current == 0;
      }

      void advance( )
      {
              if( !isPastEnd( ) )
                  current = List<Object>::cursorSpace[ current ].next:
      }

      const Object & retrieve( ) const
      {
            if( isPastEnd( ) )
                  throw BadIterator( );
            return List<Object>::cursorSpace[ current ].element;
      }
      private:
          int current;           // Current position
         friend class List<Object>;
         ListItr( int theNode ) : current( theNode ) { }
};
```

프로그람 3-17. 련결목록의 유표적인 실현에 대한 반복자

프로그람 3-18은 유표 List클라스에 대한 골격을 보여 준다. CursorNode클라스는 List

클라스의 내부에 배치되는데 이것은 동작할 때 아주 재치 있는 방법이긴 하지만 지나치게 자주 동작하지 않는다(더 상세한 내용은 제3장 제2절 2를 보시오.). 매듭들의 배렬은 cursorSpace배렬에 보관된다. 조건 ②를 모의하기 위해서는 cursorSpace배렬의 요소들에 new와 동등한것을 할당하여야 한다. 이것을 alloc산법이라고 한다. 이를 위하여 요소들이 없는 하나의 목록(빈 목록)을 만든다. 빈 목록은 요소 0을 선두매듭으로 리용한다. 그 초기구성을 표 3-4에서 보여 준다.

```
template <c1ass Object>
class ListItr;          // Incomplete declaration.

template <c1ass Object>
class List
{
public:
        List( );
        List( const List & rhs );
        ~List( );

        bool isEmpty( ) const;
        void makeEmpty( );
        ListItr<Object> zeroth( ) const;
        ListItr<Object> first( ) const;
        void insert( const Object & x, const ListItr<Object> & p);
        ListItr<Object> find( const Object & x ) const;
        ListItr<Object> findPrevious( const Object & x ) const;
        void remove( const Object & x );

public:
      struct CursorNode
      {
          CursorNode( ) : next( 0 ) { }

        private:
            CursorNode( const Object & theElement, int n ) :
                : e1ement( theElement ), next( n ) { }

            Object element;
            int       next;

            friend class List<Object>;
            friend class ListItr<Object>;
      };

      const List & operator=( const List & rhs );
private:
        int header;
```

```
        static vector<CursorNode> cursorSpace;
        static void initia1izeCursorSpace( );
        static int a11oc( );
        static void free( int p );

        friend class ListItr<Object>;
};
```

프로그람 3-18. List 에 기초한 유표의 클라스골격

표 3-4. 초기화된 cursorSpace

입구점	요 소	다음 입구점
0		1
1		2
2		3
3		4
4		5
5		6
6		7
7		8
8		9
9		10
10		0

next에서 0값은 NULL지적자와 같다. static배렬 cursorSpace는 List에 대한 모든 실례들가운데서 공통적이다. cursorSpace의 초기화는 initialize CursorSpace에서 하나의 간단한 순환으로 수행된다. static국부변수는 초기화가 한번만 실행되도록 한다. 이것을 프로그람 3-19에서 보여 준다. alloc를 실현하기 위하여 선두매듭 다음의 첫번째 요소는 빈 목록으로부터 삭제된다. delete를 실현하기 위하여 free라고 하는 산법을 서술한다. free산법을 실현하기 위하여 빈 목록의 앞에 요소를 배치한다. 프로그람 3-20은 alloc와 free의 실현을 보여 준다. 만일 리용할수 있는 공간이 없으면 론리적으로는 NULL과 같은 0을 되돌리는데 좀 더 철저한 방안으로서 new의 특징을 모방하기 위하여 례외적인것은 취급하지 않을수도 있다.

```
/**
 *  Routine to initialize the cursorSpace.
 */
template <c1ass Object>
void List<Object>::initia1izeCursorSpace( )
{
     static int cursorSpaceIsInitialized = false;
     if( !cursorSpaceIsInitialized )
     {
               cursorSpace.resize( 100 );
               for( int i = 0; i < cursorSpace.size( ); i++ )
                   cursorSpace[ i ].next = i + 1;
```

```
            cursorSpace[ cursorSpace.size( ) - 1 ].next = 0;
            cursorSpaceIsInitialized = true;
        }
    }
```

프로그람 3-19. cursorSpace 초기화

```
    /**
     * Allocate a CursorNode.
     */
    template <c1ass Object>
    int List<Object>::a11oc( )
    {
        int p = cursorSpace[ 0 ].next;
        cursorSpace[ 0 ].next = cursorSpace[ p ].next;
        return p;
    }
    /**
     *   Free a CursorNode.
     */
    template <c1ass Object>
    void List<Object>::free( int p )
    {
        cursorSpace[ p ].next = cursorSpace[ 0 ].next;
        cursorSpace[ 0 ].next = p;
    }
```

프로그람 3-20. alloc 와 free 루틴들

클라스형판들에서 정적자료성원들을 리용하는것은 정확하지 않다. 클라스형판ㄱ은 실제로 하나의 클라스가 아니므로 어떤 클라스형판의 정적자료성원은 실제로 존재하지 않는다. 실례를 들어 설명한다면 유표 List의 매개 형에 대하여 자료성원을 선언하여야 한다. 실례로 List<int>에서 cursorSpace는 다음과 같이 선언된다.

```
vector<List<int>:: CursorNode> List<int>::cursorSpace;
```

이 문법적인 표현은 아주 좋지 못하며 클라스의 주문자가 이러한 선언(내부의 구체적인 내용이 비공개부로 제공되는것)을 주어야 한다는 사실은 허용할수 없는것이다.

이것만 주면 련결목록의 유표적인 실현은 간단하다. 일치성을 보장하기 위하여 선두매듭을 가지는 목록을 실현한다. 실례로 표 3-5에서 L의 값이 5이고 M의 값이 3이면 L은 목록 a, b, e를 표시하고 M은 목록 c, d, f를 나타낸다.

표 3-5. 련결목록의 유표적인 실현에 대한 실례

입구점	요소	다음 입구점
0	-	6
1	b	9
2	f	0
3	선두매듭	7
4	-	0
5	선두매듭	10
6	-	4
7	c	8
8	d	2
9	e	0
10	a	1

련결목록의 유표적인 실현에 대한 함수들을 쓰기 위하여 앞련결실현과 같은 파라메터들을 넘기고 되돌려야 한다. 그 루틴들은 간단하다. 프로그람 3-21은 더 간단한 몇가지 산법들을 포함한다. makeEmpty와 zeroth는 제3장 제2절에서 보여 준 련결목록 판본들과 같은것이기때문에 소개하지 않는다. 프로그람 3-22에 있는 find함수는 목록에 있는 x의 위치를 되돌린다. 프로그람 3-23은 insert에 대한 유표적인 실현을 보여 준다. 삭제를 실현하기 위한 코드는 프로그람 3-24에서 보여 준다. 여기서 참조가 필요 없는 매듭은 free호출에 의하여 개선될수 있으므로 삭제된 매듭의 첨수를 기억하여야 한다. 유표적인 실현에 대한 대면부는 지적자실현에서와 같다.

```
/**
 *   Construct the list.
 */
template <c1ass Object>
List<Object>::List( )
{
    initializeCursorSpace( );
    header = a11oc( );
    cursorSpace[ header ].next = 0;
}
/**
 *   Destroy the list.
 */
template<c1ass Object>
List<Object>::~List( )
{
     makeEmpty( );
     free( header );
}
/**
 * Test if the list is logically empty.
 * Return true if empty, false otherwise.
 */
template <c1ass Object>
bool List<Object>::isEmpty( ) const
{
```

```
return cursorSpace[ header ].next == 0;
}

/**
 *   Return an iterator representing the first node in the list
 *   This operation is valid for empty lists.
 */
template <c1ass Object>
ListItr<Object> List<Object>::first( ) const
{
                    return ListItr<Object>( cursorSpace[ header ].next );
}
```

프로그람 3-21. 유표에 기초한 목록들에 대한 여러가지 간단한 루틴들

```
/**
 * Return iterator corresponding to the first node containing an item x
 *   Iterator isPastEnd if item is not found.
 */
template <c1ass Object>
ListItr<Object> List<Object>::find( const Object & x ) const
{
      int itr = cursorSpace[ header ].next;
      white( itr != 0 && cursorSpace[ itr ].element != x )
            itr = cursorSpace[ itr ].next;
      return ListItr<Object>( itr );
}
```

프로그람 3-22. 유표적인 실현에서의 find 루틴

```
/**
   * Insert item x after p.
   */
template <c1ass Object>
void List<Object>::insert( const Object & x, const ListItr<Object> & p )
{
     if( p.current != 0 )
     {
           int pos = p.current;
           int tmp = a11oc( );

           cursorSpace[ tmp ] = CursorNode( x, cursorSpace[ pos ].next );
           cursorSpace[ pos ].next = tmp;
     }
```

프로그람 3-23. 유표적인 실현에서의 련결목록에 대한 삽입루틴

```
 /**
  * Remove the first occurrence of an item x.
  */
Template <c1ass Object>
void List<Object>::remove( const Object & x )
{
      ListItr<Object> p = findPrevious( x );
      int pos = p.current;
      if( cursorSpace[ pos j.next != 0 ]
      {
            int tmp = cursorSpace[ pos ].next;
            cursorSpace[ pos ].next = cursorSpace[ tmp ].next;
            free( tmp );
      }
}
```

프로그람 3-24. 유표적인 실현에서 련결목록에 대한 삭제루틴

극히 중요한 점은 이 루틴들이 ADT설계명세서에 따른다는것이다. 그것들은 명백한 인수들을 가지고 정확한 연산들을 수행한다. 이 실현은 사용자가 알기 쉬운것이다. 유표적인 실현은 코드의 나머지에서 련결목록실현대신에 리용되게 된다. 이것은 거의나 변경할 필요가 없다. 상대적으로 find들이 적게 실행되면 기억관리루틴들의 부족으로 하여 유표적인 실현은 대단히 빨라 진다. 그러나 이 결정은 프로그람언어와 번역기에 크게 관계된다.

자유목록은 그자체의 오른쪽에서 흥미 있는 자료구조를 나타낸다. 자유목록으로부터 제거되는 요소는 free의 결과 거기에 제일 마지막으로 배치된 요소이다. 따라서 자유목록에 배치된 마지막요소는 첫번째로 제거되게 되는 요소이다. 이러한 속성을 가진 자료구조를 탄창이라고 하며 다음 절에서 고찰한다.

제3절. 탄창ADT

1. 탄창모형

탄창은 오직 top라고 하는 목록의 한 끝위치에서만 삽입과 삭제가 진행될수 있도록 제한을 가한 목록이다. 탄창에서의 기본적인 연산들은 기능이 삽입과 같은 push연산과 제일 마지막에 삽입된 요소를 삭제하는 pop연산이다. 제일 마지막에 삽입된 요소는 pop연산을 실행하기전에 top루틴을 리용하여 검사할수 있다. 일반적으로 빈 탄창에 대한 pop 또는 top는 탄창 ADT에서 오유로 처리한다. 한편 push연산을 진행할 때 공간이 부족되는 것은 실현에서의 한계이지 ADT의 오유는 아니다.

탄창들을 때때로 LIFO(Last In, First Out)목록이라고 한다. 그림 3-10에 보여 준 모형은 push들은 오직 입력만 하고 pop들과 top들은 출력만 하는 연산들이라는것을 의미한다. 빈 탄창을 만들거나 탄창이 비였는가를 검사하기 위한 일반적인 연산은 탄창의 연산범위에 속하는것들이지만 본질적으로 push와 pop로써 탄창이 할수 있는 모든것을 다 할수 있다.

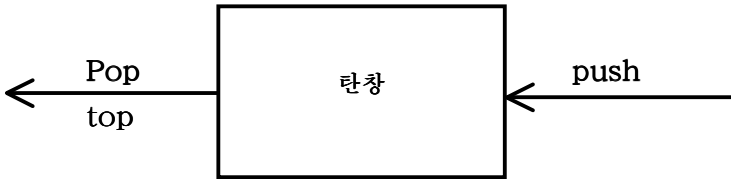



그림 3-10. 탄창모형: push 에 의해 탄창에 입력, pop 와 top 에 의해 출력

그림 3-11은 여러가지 연산들이 진행된 다음의 추상적인 탄창의 상태를 보여 준다. 일반적인 모형은 탄창의 정점에 어떤 요소가 있으며 그것은 처리될수 있는 유일한 요소라는것이다.

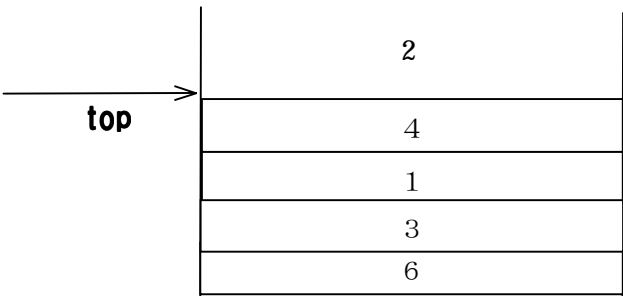



그림 3-11. 탄창모형: 접근할수 있는 유일한 정점요소

2. 탄창의 실현

탄창도 하나의 목록이므로 어떤 목록을 실현하는것과 같다. 여기서는 두가지의 일반적인 실현방법을 고찰한다. 하나는 련결목록을 리용하는것이며 다른 하나는 배렬을 리용하는것이다. 그러나 앞절에서 본것처럼 좋은 프로그람작성원리를 리용하면 호출루틴들은 리용되는 산법내용들을 알 필요가 없다.

련결목록에 의한 탄창의 실현

탄창의 첫번째 실현은 단순련결목록을 리용하는것이다. push연산을 실행하여 탄창의 앞위치에 삽입한다. pop연산을 실행하여 탄창의 앞위치에 있는 요소를 삭제할수 있다.

top연산은 단순히 목록의 앞에 있는 요소를 검사하고 그 값을 되돌린다. 때때로 top와 pop연산들은 하나로 결합된다. 앞절에서 고찰한 련결목록루틴들에 대한 호출을 리용할수 있지만 정확성을 위하여 탄창루틴들을 처음부터 다시 서술한다.

먼저 프로그람 3-25에서 클라스대면부를 준다. 여기에서는 선두매듭을 리용하지 않고 탄창을 실현한다. 클라스대면부는 비공개부에 ListNode라고 하는 하나의 struct를 포함한다. 이 struct는 class와 거의 같다. 그 차이는 무엇인가? struct는 그의 자료가 암시적으로 공개부이라는것을 제외하고는 class와 완전히 같다. 한편 class는 public표시가 나타날때까지 비공개부로 인식하고 struct는 private가 나타날 때까지 public로 처리한다.

```
template <c1ass Object>
class Stack
{
   public:
      Stack( );
      Stack( const Stack & rhs );
      -Stack( );
      bool isEmpty( ) const;
      bool isFull( ) const;
      const Object & top( ) const;
      void makeEmpty( );
      void pop( );
      void push( const Object & x );
      Object topAndPop( );
      const Stack & operator=( const Stack & rhs );
    private:
      struct ListNode
      {
          Object      element;
          ListNode *next;
          ListNode( const Object & theElement, ListNode * n = NULL )
               : element( theEletment ), next( n ) { }
      };

        ListNode *topOfStack;
 };
```

프로그람 3-25. 련결목록실현에 의한
탄창ADT의 클라스대면부

계층적인 struct의 리용은 오래동안 일반적인 C++표현형식으로 되여 왔다. ListNode의 이름은 Stack클라스의 밖으로 확장되지 않으므로 그것을 재리용할수 있도록 한다. 바꾸어 말하면 대역변수이름을 리용하지 않는다. 탄창을 제외하고 그 누구도 ListNode를 볼수 없기때문에 그의 성원들은 비공개부가 아니여도 된다.

그러나 최근의 규약변경들은 보다 더 복잡한 표현형식을 리용하려고 하면 이것이 언제나 동작하지 않는다는것을 보여 준다. 개별적으로 서술된 산법이 계층적인 클라스(또

는 그에 대한 지적자)를 되돌려야 할 필요가 있을 때 되돌림형이 대역적인 범위에 속하기때문에 일반적으로 하나의 문제가 발생한다.

실례로 탄창에서 제일 바닥에 있는 매듭에 대한 지적자를 되돌리는 내부적인 비공개산법을 서술하려고 한다고 하자. 이것은 대면부에서 다음과 같이 쓸수 있다.

```
ListNode *   getBottomNode( ) const;
```

이것을 대면부와 분리하여 실현하면

```
template <class Object>
List<Object>:: ListNode * ListNode<Object>::getBottomNode( ) const
{
        …
}
```

을 얻는다.

이것은 List<object>::ListNode표현이 대역적인 범위(또는 Borland 5.0에서 처리하는것처럼)로 되지만 ListNode는 비공개부이기때문에 허용될수 없다. 이것은 C++에서 일반적으로 제기되는 문제를 설명한다. 즉 여러가지 특징들이 동시에 리용되면 그때 충돌이 발생한다. 지어 계층적인 구조체들이 정확하게 작성되였을 때조차 일부 번역기들(실례로 g++와 같은)은 모든 단계들을 정확히 분석하지 않는다. Stack클라스는 이 표현형식이 동작하는 적은 경우들중의 하나이다.

구축자를 포함하여 여러가지 소소한 루틴들을 프로그람 3-26에서 실현한다. 여기에서 탄창은 결코 다 차지 않는다는것에 주의하여야 한다.

```
/**
 * Construct the stack.
 */
template <c1ass Object>
Stack<Object>::Stack( )
{
        topOfStack = NULL;
}
/**
 * Destructor.
 */
template <c1ass Object>
Stack<Object>::~Stack( )
{
        makeEmpty( );
}
/**
 * Test if the stack is logically fu11.
 * Return false always, in this implementation
```

```
*/
template <c1ass Object>
bool Stack<Object>::isFu11( ) const
{
        return false;
}

/**
  * Test if the stack is logically empty.
  * Return true if empty, false otherwise.
  */
template <c1ass Object>
bool Stack<Object>::isEmpty( ) const
{
     return topOfStack == NULL;
}

/**
  * Make the stack logically empty.
  */
template <c1ass Object>
void Stack<Object>::makeEmpty( )
{
       white( !isEmpty( ) )
               pop( );
}
```

프로그램 3-26. 련결목록탄창실현에 대한 몇가지 간단한 산법들

련결목록의 앞에 삽입하기 위하여 push를 실현하는데 거기에서 목록의 앞은 탄창의 정점으로서 봉사한다(프로그람 3-27). 목록의 첫 위치에 있는 요소를 검사하기 위하여 top를 실행한다(프로그람 3-28). 목록의 앞에 있는 요소를 삭제하기 위하여 pop를 실현하는데 이것은 간단히 topOfStack를 전진시키는 방법으로 한다(프로그람 3-29). pop는 탄창이 비면 제외하고 프로그람 3-30에서 보는것처럼 topAndPop는 삭제된 항목의 복사를 되돌린다.

```
/**
   * Insert x into the stack.
   */
 template <c1ass Object>
void Stack<Object>::push( const Object & x )
{
      topOfStack = new ListNode( x, topOfStack );
}
```

프로그람 3-27. 련결목록으로 실현한 탄창에 요소를 넣기 위한 루틴

```
/**
 * Get the most recently inserted item in the stack.
 * Return the most recently inserted item in the stack
 * or throw an exception if empty.
 */
template <c1ass Object>
const Object & Stack<Object>::top( ) const
{
    if( isEmpty( ) )
        throw Underflow( );
    return topOfStack->e1ement;
}
```

프로그람 3-28. 련결목록으로 실현된 탄창에 있는 정점요소를 되돌리는 루틴

```
/**
 *  Remove the most recently inserted item from the stack.
 *  Throw the Underflow exception if the stack is empty.
 */
template <c1ass Object>
void Stack<Object>::pop( )
{
    if( isEmpty( ) )
        throw new Underf1ow( );

    ListNode *oldTop = topOfStack;
    topOfStack = topOfStack->next;
    delete oldTop;
}
```

프로그람 3-29. 련결목록으로 실현한 탄창으로부터 요소를 뽑기 위한 루틴

```
/**
 * Return and remove the most recently inserted item from the stack
 * Throw the Underflow exception if the stack is empty.
 */
template <c1ass Object>
Object Stack<Object>::topAndPop( )
{
    Object topItem = top( );
    pop( );
    return topItem;
}
```

프로그람 3-30. 련결목록으로 실현된 탄창에서 요소를 뽑아서 되돌려 주는 루틴

모든 연산들을 진행하는데 상수시간이 걸린다는것은 명백하다. 왜냐하면 임의의 루틴들의 그 어디에도 탄창크기에 대한 참조(빈 탄창들은 제외)조차 없기때문이다. 이 상수시간은 훨씬 더 작은 순환에 해당되며 그것은 탄창크기에 관계된다. 이 실현의 결함은 일부 언어들에서는 new에 대한 호출들이 시간이 많이 걸릴수도 있다는것이다. 특히 간단한 련결조작들에 비해 보면 더 잘 알수 있다. 이 몇가지는 두번째 탄창을 리용하여 피할수 있는데 그것은 초기에 빈 탄창이다. 어떤 요소가 첫번째 탄창으로부터 떨어 져 나올 때 그것은 단순히 두번째 탄창에 배치된다. 그다음에 첫번째 탄창에 새로운 요소들이 필요되면 두번째 탄창을 먼저 검사한다.

프로그람 3-31에서 복사대입연산자를 보여 주는데 일반적인 별명검사와 *this의 되돌림을 포함한다. 복사하기전에 이미 탄창에 할당되였던 기억기의 류실을 피하기 위하여 현재 탄창을 비여 놓는다. 빈 목록에 첫번째 매듭을 만들고 새로운 ListNode를 추가하면서 목적하는 목록의 끝까지 rhs를 따라 나간다.

복사구축자를 실현하기 위하여 topOfStack위치를 NULL로 만든 다음에 복사대입연산자를 호출한다. 이것을 프로그람 3-31에서 보여 준다.

```
/**
 * Deep copy.
 */
template <c1ass Object>
const Stack<Object> & Stack<Object>::
operator=( const Stack<Object> & rhs )
{
        if( this != &rhs )
        {
            makeEmpty( );
            if( rhs.isEmpty( ) )
                return *this;

            ListNode *rptr = rhs.topOfStack;
            ListNode *ptr = new ListNode( rptr->e1ement );
            topOfStack = ptr;
            for( rptr = rptr->next; rptr != NULL; rptr = rptr->next )
                    ptr = ptr->next = new ListNode( rptr->e1ement );
        }
        return *this;
}
/**
 * Copy constructor.
 */
template <c1ass Object>
Stack<Object>::Stack( const Stack<Object> & rhs )
{
```

```
        topOfStack = NULL;
        *this = rhs;
    }
```

프로그람 3-31. 련결목록에 기초한 탄창에서 복사대입연산자와 복사구축자

배렬에 의한 탄창의 실현

또 하나의 실현은 련결목록을 리용하지 않는 보다 더 일반적인 해결방법이다. 이 방법에서 유일하게 잠재적인 위험은 배렬크기를 미리 선언하여야 한다는것이다. 이것은 일반적으로 아주 많은 탄창연산들이 있을 때에도 어떤 시점에서 탄창의 실제 요소들의 수는 그리 크지 않으므로 전형적인 응용들에서 문제로 되지 않는다. 더우기 많은 공간을 소비하지 않고 배렬을 충분히 크도록 선언하는것은 일반적으로 쉽다. 이것이 가능하지 못하면 련결목록실현이나 련습문제 3-29에서 준것과 같이 탄창용량을 동적으로 확장하는 수법을 리용할수 있다.

배렬실현을 리용하면 그 실현은 어렵지 않다. 매개의 탄창과 관련된것은 theArray와 topOfStack인데 그것은 빈 탄창인 때 -1로 된다(이것이 빈탄창이 초기화되는 방법이다.). 탄창에 어떤 요소 x를 넣기 위하여서는 topOfStack를 증가시키고 그다음 theArray[topOfStack] =x로 설정한다. 탄창으로부터 정점요소를 뽑기 위해서는 되돌림값을 theArray[topOfStack]로 설정하고 그다음 topOfStack를 감소시킨다.

이 연산들은 상수시간뿐아니라 대단히 빠른 상수시간에 실행된다는데 주의를 돌려야 한다. 일부 기계들에서 push와 pop들은 자동적으로 증가, 감소하는 주소를 가진 등록기에 대한 연산을 수행하는 하나의 기계명령으로 서술할수 있다. 가장 현대적인 기계들이 지령모임부분에 탄창연산들을 가진다는 사실은 콤퓨터과학에서 탄창이 배렬 다음으로 가장 기초적인 자료구조라는것을 강조한다.

탄창실현의 효과성에 영향을 미치는 하나의 문제는 오유를 검사하는것이다. 련결목록실현은 오유들을 주의깊게 검사하였다. 우에서 서술한것처럼 빈 탄창에 대한 pop나 탄창자리넘침에 대한 push는 배렬한계를 초과시키며 연산파괴를 일으킨다. 이것은 명백히 허용할수 없는것이지만 만일 이 조건들에 대한 검사들이 배렬실현에 부과되면 그것들은 실제적인 탄창조작만큼 많은 시간이 걸리게 된다. 이러한 리유로 탄창루틴들에서는 일반적인 실천에서 오유수정이 극히 중요한곳을 제외하고 오유검사를 하지 않는다(조작체계에서처럼). 대체로 많은 경우 자리넘침이 생기지 않도록 탄창을 충분히 크게 선언하거나 pop를 리용하는 루틴들이 빈 탄창을 pop하지 않도록 담보하는 방법으로 이러한 현상을 없앨수 있다. 그러나 이것은 잘해야 겨우 동작하는 코드로 될수 있으며 특히 프로그람들이 너무 커서 그것을 여러명에 의해서 또는 여러번에 걸쳐 만들어 질 때 더하다. 탄창연

산들이 그렇게 빠른 상수시간을 가지므로 프로그람실행시간의 중요한 부분은 이 루틴들에서 소비된다. 이것은 일반적으로 오유검사를 없앨수 없다는것을 의미한다. 사용자는 항상 오유검사를 하여야 하는데 만일 그것들이 너무 많으면 실제로 시간이 지나치게 소비되는 검사들에만 설명문을 줄수 있다. 이 모든것을 종합하여 배렬을 리용하여 일반적인 탄창을 실현하는 루틴을 쓸수 있다.

Stack클라스대면부는 프로그람 3-32에서 보여 주었다.

```
template <c1ass Object>
class Stack
{
    public:
        explicit Stack( int capacity = 10 )
        bool isEmpty( ) const;
        bool isFull( ) const;
        const Object & top( ) const;
        void makeEmpty( );
        void pop( );
        void push( const Object & x );
        Object topAndPop( );
    private:
        vector<Object> theArray;
        int            topOfStack ,
};
```

프로그람 3-32. 배렬실현에 대한 Stack 클라스대면부

탄창루틴들의 실현은 프로그람 3-33 ~ 3-38과 같이 아주 간단하고도 정확하게 서술할수 있다.

이 클라스는 몇가지 미묘한 문제들을 가진다. 우선 좋은 자료가 있다. 즉 자료성원들이 vector와 int형이고 《3대기능》들이 원만히 정의되였기때문에 중요하게 쓰이는 해체자들, 복사구축자들, 복사대입연산자들이 Stack에 알맞게 자동적으로 정의된다. 여기서 특별한 처리는 하지 말아야 한다. 이것은 자체의 방식대로 처리하는 C++를 설명한다.

여기에 몇가지 흥미 있는 구체적인 수법들이 있다.

① top는 객체를 상수참조로 되돌린다. 그러나 top와 pop는 값에 의한 되돌림을 리용한다. 왜 차이가 있는가? 그 차이는 제1장 제5절 3에서 설명하였다. top가 접근자이므로 되돌려 지는 값은 여전히 탄창에 있게 된다. 따라서 상수참조에 의한 되돌림을 리용할수 있다. topAndPop에서 되돌림값은 론리적으로 탄창으로부터 제거되므로 그것은 더이상 존재하지 않는다. 더우기 상수참조에 의한 되돌림은 적당하지 않다. 사실 배렬에 기초한 실현에서 topOfStack만을 변경하여 정점

항목이 있는 배렬입구점의 내용들이 변경되지는 않는다. 그러므로 상수참조에 의한 되돌림이 기술적으로 담보된다. 그러나 그것은 좋지 못한 방안으로 된다.

② 일반적으로 처음 배우는 사람들의 실책은 클라스대면부에서 그의 선언위치에 배렬의 크기를 포함하는것이다. 이것은 잘못된것이다. 자료성원들은 단지 그것들의 형들만 표시할수 있다. 초기화는 구축자에서 실행하여야 한다. 이에 대한 가장 편리한 방법은 그림 3-4에서 보여 준것처럼 초기화자표를 리용하는것이다. 좀 더 불충분한 방법은 초기화자표를 리용하지 않고 그대신에 구축자의 본체에서 resize를 호출하는것이다.

```
/**
* Construct the stack.
*/
template <c1ass Object>
Stack<Object>::Stack( int capacity ) : theArray( capacity )
{
        topOfStack = -1;
}
```

프로그람 3-33. 배렬로 실현한 탄창구축

```
/**
  *   Test if the stack is logically empty.
  *   Return true if empty, false otherwise.
  */
 template <c1ass Object>
 bool Stack<Object>::isEmpty( ) const
 {
         return topOfStack == -1;
 }
/**
 * Test if the stack is logically full.
 * Return true if full, false otherwise.
 */
 template <c1ass Object>
 bool Stack<Object>::isFull( ) const
 {
                return topOfStack == theArray.size( ) - 1;
 }

 /**
  * Make the stack logically empty.
  */
template <c1ass Object>
```

```
void Stack<Object>::makeEmpty( )
{
        topOfStack = -1;
}
```

프로그람 3-34. 배렬을 실현한 한개 행으로 된 루틴들

```
/**
* Insert x into the stack, if not already fu11.
* Throw the Overflow exception if the stack is already full.
*/
template <c1ass Object>
void Stack<Object>::push( const Object & x )
{
        if( isFu11( ) )
                throw Overf1ow( ) ;
        theArray[ ++topOfStack ] = x;
}
```

프로그람 3-35. 배렬로 실현한 탄창에
넣기 위한 루틴

```
/**
 * Get the most recently inserted item in the stack.
 * Does not alter the stack.
 * Return the most recently inserted item in the stack.
 * Throw the Underflow exception if the stack is already empty.
 */
template <c1ass Object>
const Object & Stack<Object>::top( ) const
{
        if( isEmpty( ) )
                throw Underf1ow( );
        return theArray[ topOfStack ];
}
```

프로그람 3-36. 배렬로 실현한 탄창의
정점요소를 되돌리는 루틴

```
/**
 * Remove the most recently inserted item from the stack.
 * Throw the Underflow exception if the stack is already empty.
 */
template <c1ass Object>
voidStack<Object>::pop( )
```

```
    {
        if( isEmpty( ) )
            throw new Underf1ow( );
        topOfStack--;
    }
```

프로그람 3-37. 배렬로 실현한 탄창에서 요소를 꺼내는 루틴

```
/**
 * Return and remove the most recently inserted item from the stack.
 * Return the most recently inserted item.
 * Throw the Underflow exception if the stack is already empty.
 */
template <c1ass Object>
Object Stack<Object>::topAndPop( )
{
        if( isEmpty( ) )
            throw Underflow( );
        return theArray[ topOfStack-- ];
}
```

프로그람 3-38. 배렬로 실현된 탄창에서
정점요소를 꺼내서 되돌리는 루틴

3. 응용

만일 어떤 목록에 할당된 연산들을 제한하면 이 연산들이 아주 빨리 실행될수 있다는것은 놀랍지 않은것으로 되여 왔다. 그러나 대단히 놀라운것은 적은 수의 연산들이 아주 강력하고 중요하다는것이다. 여기서는 탄창에 대한 많은 응용들가운데서 세가지를 고찰한다. 세번째 응용에서는 프로그람들을 어떻게 조직하는가를 구체적으로 준다.

기호들의 균형잡기

번역기들은 사용자의 프로그람에 문법적인 오유가 있는가를 검사한다. 그러나 종종 기호 하나가 부족해서(큰 괄호나 주해시작기호를 놓치는것과 같은) 번역기가 실제 오유는 식별하지 못하고 많은 행을 검사하게 되는 결과를 발생시킨다.

이 경우에 쓸모가 있는 도구는 모든 기호들이 균형이 잡혔는가를 검사하는 프로그람이다. 따라서 모든 오른쪽 대괄호와 꺽인괄호, 소괄호들은 그의 왼쪽 기호들과 짝을 맞추도록 대응되여야 한다. [()]의 순서는 옳은 표시지만 [(])는 잘못된 표시이다. 그것은 명백히 거대한 프로그람을 서술하는데서 상당히 좋지 못한것이지만 이러한 오유를 검사하는것은 그리 힘들지 않다. 그것은 소괄호, 꺽인괄호, 대괄호들이 짝을 맞추었는가를 검

사하고 나타난 어떤 다른 문자를 간단히 무시해 버리는것이다.

간단한 알고리듬은 탄창을 리용하여 다음과 같이 서술할수 있다.

빈 탄창을 만든다. 파일의 끝까지 문자들을 읽는다. 만일 그 문자가 열린기호이면 그것을 탄창에 넣는다. 만일 닫긴기호이고 그때 탄창이 비였으면 오유를 내보내고 비지 않았으면 탄창에서 뽑는다. 만일 뽑아 낸 기호가 열린 기호에 대응하지 않으면 그때 오유를 내보낸다. 만일 파일의 끝에서도 탄창이 비지 않았다면 오유를 내보낸다.

사용자는 이 알고리듬의 동작을 확신할수 있다. 이 알고리듬은 정확히 선형적이며 실제로 입력에 대하여 단지 하나의 단계로서 수행된다. 따라서 그것은 직결방법이며 아주 빨리 처리된다. 기타 처리를 진행하여 오유가 발생할 때 무엇을 할것인가를 결정할수 있다. 실례로 적당한 원인을 밝혀 주는것을 들수 있다.

뒤배치식

전자수판을 가지고 상점에서 산 물건들의 값을 계산한다고 하자. 이를 위하여 수값들을 더하고 그 결과에 1.06을 곱하는데 이것은 부파세를 고려한 어떤 물건들의 판매가격을 계산한것이다. 만일 판매가격들이 4.99, 5.99, 6.99이면 이것들을 입력하는 일반적인 방법은 다음의 순서로 표시된다.

4.99 + 5.99 + 6.99 * 1.06 =

수판에 따라 이것은 19.05라는 고의적인 결과라든가 18.39라는 과학적인 결과를 나타낸다. 대부분의 간단한 4칙계산수판들은 첫번째 결과를 주지만 대부분의 고급한 콤퓨터들은 곱하기가 더하기보다 더 높은 우선권을 가지고 먼저 처리된다.

한편 어떤 물건들은 부파세가 붙을수도 있고 붙지 않을수도 있다. 따라서 만일 첫번째와 마지막물건들만 부파세가 붙는다면 그 표시순서

4.99 * 1.06 + 5.99 + 6.99 * 1.06 =

는 과학적인 계산기에서는 정확한 결과(18.69)를 주고 간단한 수판들은 틀린 결과(19.37)를 줄것이다. 과학적인 계산기는 일반적으로 소괄호를 처리할수 있으며 따라서 그것을 리용하여 항상 옳은 결과를 얻을수 있지만 간단한 수판은 중간결과를 기억해야 한다.

이 실례에서 전형적인 계산순서는 먼저 4.99와 1.06을 곱하고 이 결과를 A_1에 기억시켜야 한다. 그다음 5.99와 A_1을 더하고 그 결과를 A_1에 보관한다. 6.99와 1.06을 곱하고 그 결과를 A_2에 보관하고 A_1과 A_2를 더하는것으로 끝내여 마지막결과를 A_1에 보관한다. 이 연산순서를 다음과 같이 쓸수 있다.

4.99 1.06 * 5.99 + 6.99 1.06 * +

이 표기법을 뒤배치식(postfix) 또는 **역뽈스까표기법**(*reverse Polish notation*)이라고 하는데

우에서 서술한것처럼 정확히 평가된다. 이것을 계산하기 위한 가장 간단한 방법은 탄창을 리용하는것이다. 수자가 입력되면 그것을 탄창에 넣고 연산자가 입력되면 그것을 탄창에서 꺼낸 두 수값(기호)들에 적용하고 그 결과를 탄창에 넣는다. 실례로 뒤배치식

6 5 2 3 + 8 * + 3 + *

는 다음과 같이 계산된다. 즉 첫 4개의 기호(수자)는 탄창에 넣어 진다. 탄창의 결과는 그림 3-12 ㄱ와 같다. 다음 《+》가 읽어 지므로 탄창에서 3과 2를 꺼내서 그것들을 더한 합 5를 넣는다(그림 3-12 ㄴ). 다음 8이 넣어 진다(그림 3-12 ㄷ). 다음 《* 》이 읽어 지므로 8과 5를 꺼내서 5*8=40을 넣는다(그림 3-12 ㄹ). 다음 《+》이므로 40과 5를 꺼내서 5 + 40 = 45를 넣는다(그림 3-12 ㅁ). 다음 3을 넣는다(그림 3-12 ㅂ). 그리고 《+》이므로 3과 45를 꺼내고 45 + 3 = 48을 넣는다(그림 3-12 ㅅ). 마지막으로 《*》가 나타나므로 48과 6을 꺼내서 그 결과인 6 * 48 = 288을 넣는다(그림 3-12 ㅇ).

그림 3-12. 탄창결과

뒤배치식의 계산시간은 $O(N)$인데 그것은 입력하는 매개 요소에 대한 처리가 탄창연산들로 되여 있으므로 상수시간을 가지기때문이다. 이것을 처리하기 위한 알고리듬은 간단하다. 어떤 식이 뒤배치로 주어 졌을 때 어떤 우선권에 대한 규칙을 알 필요가 없는데 이것이 이 표현법의 우점이다.

사이배치를 뒤배치로 변환

탄창은 뒤배치식을 계산하는데 리용할수 있을뿐아니라 표준형태의 어떤 표현(infix와 같은)을 뒤배치식으로 변환할수 있다. +, *, (,)연산자들만을 가지고 보통의 우선권규칙을 리용하여 일반적인 수값연산문제를 해결할수 있다. 더 나아가서 그 표현이 옳다는것을 알게 된다. 사이배치식

$$a + b * c + (d * e + f) * g$$

을 뒤배치로 변환해 보자. 이때의 정확한 결과는 a b c * + d e * f + g * +이다.

피연산수가 읽어 지면 그것을 즉시 출력한다. 연산자들은 그 즉시 출력하지 않으며 따라서 그것들은 어떤 장소에 보관되여야 한다. 수행해야 할 정확한 처리는 읽어 진 연산자들을 출력하지 않고 탄창에 배치하는것이다. 초기에 빈 탄창을 가지고 시작한다.

만일 오른쪽 소괄호가 나타나면 대응하는 왼쪽 소괄호가 나타날 때까지 기호들을 써내려 가면서 탄창에서 뽑아 내는데 왼쪽 소괄호는 출력되지 않고 꺼내기만 한다.

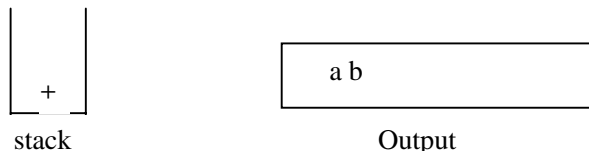
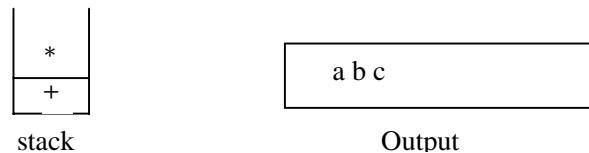
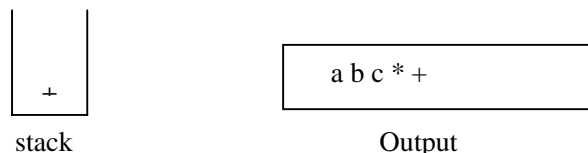
만일 어떤 다른 기호들 즉 +, *, (와 같은 기호들이 나타나면 더 낮은 우선권을 가진 입구점들이 나타날 때까지 탄창으로부터 입구점들을 뽑는다. 한가지 례외는)기호를 처리할 때를 제외하고 탄창으로부터 (기호를 결코 제거하지 않다는것이다. 이 연산수행에서 +는 가장 낮은 우선권을 가지고 (기호는 가장 높은 우선권을 가진다. 뽑기가 수행되면 그 연산자를 탄창에 넣는다.

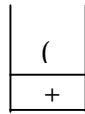
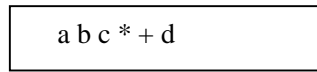
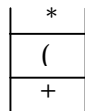
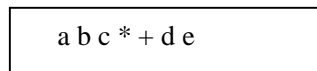
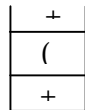
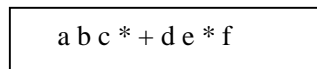
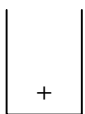
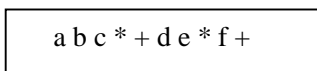
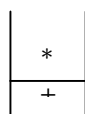
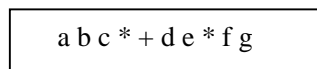
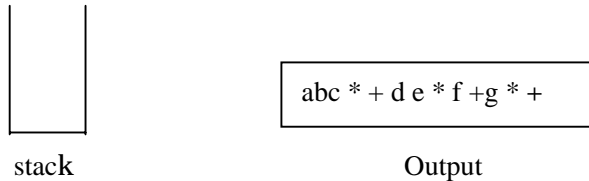
마지막으로 입력의 끝을 읽으면 빈 탄창이 될 때까지 탄창에서 꺼내여 출력상에 기호들을 써나간다.

이 알고리듬의 중심내용은 연산자가 나타나면 그것을 탄창에 배치한다는것이다. 탄창은 필요한 연산자들을 나타낸다. 그러나 높은 우선권을 가지고 탄창에 보관된 어떤 연산자들은 해당 연산이 끝나면 더는 필요 없으므로 탄창에서 꺼내 진다. 따라서 탄창에 연산자들을 배치하기전에 탄창에 있는 연산자들과 현재 연산자보다 앞서 실행된 연산자가 꺼내 진다. 이것은 다음의 표로 설명할수 있다.

표현	세번째 연산자가 처리된 때의 탄창	동작
a*b-c+d	−	−가 실행되고 +가 넣어 진다.
a/b+c*d	+	아무런 처리도 진행되지 않고 *가 넣어 진다
a-b*c/d	− *	*가 실행되고 /가 넣어 진다.
a-b*c+d	− *	*와 −가 실행되고 +가 넣어 진다.

소괄호는 단순히 추가적인 복잡성을 더해 준다. 왼쪽 소괄호는 그것이 입력되는것이면 높은 우선권을 가진 연산자로(필요한 연산자로 남아 있게 하기 위해서), 그것이 탄창에 있는것이면 낮은 우선권을 가진 연산자로 고찰할수 있다(그것이 어떤 연산자에 의해서 제거되지 않도록 하기 위해서). 오른쪽 소괄호들은 특별한 경우서 취급된다.

이 알고리듬이 어떻게 실행하는가를 보기 위하여 사이배치식을 뒤배치식으로 변환시켜 보자. 먼저 a기호가 읽어 지면 그것을 출력한다. 그다음 +가 읽어 지면 탄창에 넣는다. 다음의 b는 출력한다. 이 경우에 사건들의 상태는 다음과 같다.

다음에 *가 읽어 진다. 연산자탄창에서 정점입구점은 *보다 낮은 우선권을 가지므로 출력되지 않고 탄창에 보관된다. 다음에 C가 읽어 지고 출력된다. 지금까지는

이다. 다음 기호는 +이다. 탄창을 검사하고 *를 뽑아서 그것을 출력한다. 다음에 뽑는 것이 +인데 이것은 새로 읽어 진것과 우선권이 같지만 더 낮지는 않으므로 그것을 출력하고 새로 읽어 진 +를 탄창에 넣는다.

다음에 읽어 진 기호는 (인데 가장 높은 우선권을 가지므로 탄창에 배치한다. 그다

음 d를 읽어서 출력한다.

다음에 *를 계속 읽는다. 열린 소괄호는 닫긴 소괄호가 입력될 때 삭제되기때문에 출력하지 않는다. 다음 e를 읽어서 출력한다.

다음 기호는 +이므로 *를 뽑아서 출력하고 +를 탄창에 넣는다. 그다음 f를 읽고 출력한다.

이제)를 읽어 들이는데 따라서 탄창에서 (의 뒤부분이 비게 된다. 그때 +를 출력한다.

다음에 읽어 들이는 *는 탄창에 넣어 진다. 그다음 g를 읽어서 탄창에 보낸다.

입력이 끝나면 탄창이 빌 때까지 탄창으로부터 기호들을 뽑아서 출력한다.

앞에서와 같이 이 변환은 다만 $O(N)$의 시간을 요구하며 입력자료들에서 하나의 단계로써 실행된다. 이러한 방법으로 더하기와 우선권이 같은 덜기, 곱하기와 우선권이 같은 나누기를 넣어서 덜기와 나누기연산들도 처리할수 있다. 문제는 a-b-c표현이 abc- -가 아니라 ab-c-로 변환된다는것이다. 이 알고리듬은 연산자들이 왼쪽에서 오른쪽으로 배치되기때문에 정확히 처리된다. 제곱은 오른쪽에서 왼쪽으로 계산되기때문에 이 알고리듬이 일반적으로 반드시 그렇게 되는것은 아니다. 즉 $2^{2^3}=2^8=256$으로 계산되여야지 $4^3= 64$로 되는것은 아니다. 제곱연산을 처리하는 방법은 련습문제에서 주었다.

함수호출

균형 잡힌 기호들을 검사하는 알고리듬은 번역된 수속형언어와 대상지향언어들에서 함수호출을 실현하는 방법을 제시한다. 여기서 문제로 되는것은 새로운 함수를 호출할 때 호출루틴들에 대한 국부변수들은 체계에 보관되여야 하는것이다. 만일 그렇게 하지 않으면 새로운 함수가 호출루틴의 변수들에 의해 리용되는 기억기를 겹쳐 사용할수 있기때문이다. 더우기 새로운 함수가 처리된 다음에 되돌아 가야 할 위치를 알수 있도록 루틴에서의 현재 위치가 보관되여야 한다. 변수들은 일반적으로 번역기에 의해 기계등록기들에 할당(보통 모든 함수들은 일부 변수들이 등록기 #1에 할당되게 한다.)되며 특히 재귀가 포함된다면 반드시 충돌이 생기게 된다. 이 문제가 기호들의 균형맞추기문제와 류사한 리유는 함수호출과 함수되돌림이 본질적으로는 열린소괄호와 닫긴소괄호에 대한 문제와 같은것이기때문이다.

함수를 호출할 때 변수이름들에 대응하는 등록기값들과 그리고 전형적으로 등록기에 존재하여 프로그람계수기로부터 얻어 질수 있는 되돌림주소와 같이 중요한 정보는 추상적으로 한토막의 기억기에 보관되여 더미의 정점에 넣어 진다. 그때 조종은 새로운 함수에로 넘어 가는데 이것은 자기의 값들을 가지고 등록기들을 배치할수 있도록 등록기들을 해방한다. 다른 함수들이 호출되여도 역시 같은 처리를 진행한다. 함수가 되돌려 질 때 그것은 더미의 정점에 있는 ≪기록≫에 나타나고 모든 등록기들이 수복된다. 그다음 되돌림을 뛰여 넘는다.

정확히 이 모든 처리는 탄창을 리용하여 진행할수 있으며 대체로 재귀를 실현하는 모든 프로그람작성언어들에서 발생된다. 보관된 정보를 능동기록 또는 탄창틀이라고 한다. 형태적으로 약간의 조정이 이루어 지는데 즉 현재의 환경이 탄창의 정점에 표현된다. 따라서 되돌림은 복사하지 않고 전단계의 환경을 준다. 실제 콤퓨터에서 탄창은 사용자

기억기의 웃쪽에서부터 아래쪽으로 주기적으로 증가되며 많은 체계들에서는 탄창의 자리넘침에 대한 검사를 하지 않는다. 탄창에서는 너무 많은 함수들이 동시에 처리되기때문에 자료들이 탄창공간의 밖으로 넘쳐 날 가능성이 존재한다. 탄창공간의 부족은 치명적인 오유를 가져 온다.

탄창의 자리넘침을 검사하지 않는 언어들과 체계들에서 프로그람들은 명백한 설명이 없이 파괴된다.

정상적인 경우 사용자에게는 탄창공간이 부족되지 않는데 탄창공간이 부족되는 현상은 보통 기초조건을 무시하는 경우의 무한재귀에서 나타난다. 한편 보기에는 결함이 없는것 같은 어떤 프로그람들에서도 탄창공간이 부족할수 있다. 프로그람 3-39의 루틴은 어떤 매듭에서 시작하여 련결목록을 출력하는것인데 실제로 정확하다. 그것은 빈 목록인 때의 기초조건을 적당히 조종하며 재귀처리도 좋다. 이 프로그람은 정확하다. 그러나 만일 목록이 20,000개의 요소를 출력한다면 3행에 대한 계층적인 호출을 표현하는 20,000개의 능동기록들에 대한 탄창이 있어야 한다. 능동기록들은 그것들이 포함하는 모든 정보로 하여 전형적으로 커진다. 따라서 이 프로그람들은 대체로 탄창공간의 부족현상을 가져 온다(만일 20,000개의 요소들이 프로그람을 만드는데 충분하지 못하면 더 큰 탄창에 수값들을 다시 배치하여야 한다).

```
          /**
          * Print List from ListNode p onwards; assume friendship granted.
          */
          template <c1ass Object>
          void printList( ListNode<Object> *p )
          {
/*1*/             if( p == NULL )
/*2*/                 return;
/*3*/           cout « p->e1ement « endl ;
                printList( p->next );
          }
```

프로그람 3-39. 련결목록출력에서 재귀의 좋지 못한 리용

이 프로그람은 **꼬리재귀**(*tail recursion*)로 알려 진 나쁜 재귀방법의 리용에 대한 한가지 실례이다. 계산한정재귀는 마지막행에서의 재귀호출에 기인된다. 계산한정재귀는 하나의 while순환고리에 재귀함수의 본체를 포함하고 4행에서의 재귀호출을 하나의 대입문으로 치환하여 기계적으로 없앨수 있다. 이것은 그 어떤 내용도 보관할 필요가 없는것으로서 우에서의 재귀호출을 모의할수 있는데 재귀호출이 완료된후에도 실제로 보관된 값을 알 필요가 없다. 이러한 리유로 재귀호출에 리용되였던 값들을 가지고 함수의 첫 부분으로 갈수 있다. 프로그람 3-40의 함수는 이 알고리듬에 의하여 개선된 함수이다. 계

산한정재귀의 제거는 어떤 번역기들이 그것을 자동적으로 수행할수 있을만큼 아주 간단하다. 그렇지만 그것은 사용자가 요구하지 않은 이상 그렇게 하지 않는것이 좋다.

```
/**
* Print List from ListNode p onwards; assume friendship granted.
*/
template <c1ass Object>
void printList( ListNode<Object> *p )
{
        whi1e( true )
        {
          if( p == NULL )
                return;
                cout << p->e1ement << end1;
                p = p->next;
        }
}
```

프로그람 3-40. 재귀를 리용하지 않고 목록을 출력, 번역기는 사용자가 처리하지 않아도 이렇게 변환할수 있다.

재귀는 언제나 완전히 없앨수 있지만(번역기들은 그것을 기호언어로 변환하여 그렇게 할수 있다.) 그 실현은 아주 지루한것이다. 일반적인 방법은 탄창을 리용하는것이며 이것은 탄창에서 제공된 최소값을 넣는것을 관리할수 있을 때에만 가치가 있다. 여기서는 비재귀프로그람들이 동등한 재귀프로그람들보다 일반적으로 더 빠르다고 하여도 속도측면에서의 우점이 재귀의 제거로부터 얻어 지는 명백하지 못한 결과를 허용하게 한다는데 대하여 더이상 전개하지 않는다.

제4절. 대기렬ADT

탄창과 같이 대기렬도 목록이다. 그러나 대기렬에서는 한쪽 끝에서 삽입만 진행되고 다른 끝에서는 삭제만 진행된다.

1. 대기렬모형

대기렬에서 **기본연산**(basic operation)들은 꼬리라고 하는 목록의 끝에 요소를 삽입하는 enqueue연산과 목록의 선두라고 하는 시작위치의 요소를 삭제(동시에 그 요소를 되돌리는)하는 dequeue연산이다. 그림 3-13에 대기렬의 추상적인 모형을 보여 준다.

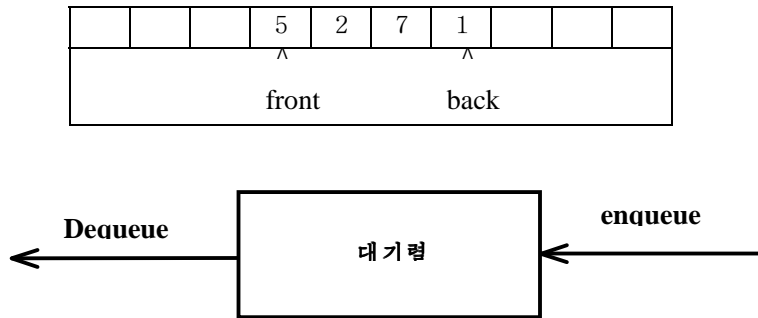



그림 3-13. 대기렬모형

2. 배렬에 의한 대기렬의 실현

탄창과 마찬가지로 어떤 목록을 대기렬에 알맞게 실현할수 있다. 탄창과 류사하게 련결목록과 배렬에 의한 대기렬실현들은 매개 연산에 대하여 빠른 $O(1)$실행시간을 제공한다. 련결목록실현은 간단하므로 련습문제로 남겨 둔다. 여기에서는 대기렬에 대한 배렬실현을 고찰하기로 한다.

매 대기렬자료구조는 배렬 theArray와 대기렬의 끝들을 나타내는 front와 back위치들을 설정한다. 또한 대기렬에서 실지 요소들의 수 currentSize를 기록해 둔다. 다음의 표는 어떤 대기렬의 내부상태를 보여 준다.

대기렬연산들은 명백하다. 요소 x를 대기렬에 넣는 enqueue연산에서는 currentSize와 back를 증가시키고 theArray[back] = x로 설정한다. 요소를 삭제하는 연산 dequeue는 되돌림값을 theArray[front]로 설정하고 currentSize를 감소시키며 그다음 front를 증가시킨다. 다른 산법들도 가능하다(이것은 뒤에서 고찰함). 여기서는 오유검사에 대해서 설명하기로 한다.

대기렬에 대한 배렬실현에는 하나의 중요한 문제가 있다. enqueue연산을 10번 수행한 다음에 대기렬은 자리가 다 차게 된다. 때문에 그때 back는 마지막배렬첨수로 되며 다음번 enqueue연산은 존재하지 않는 위치에서 진행되게 된다. 그러나 이미 여러개의 요소들이 삭제되였다면 대기렬에는 적은 수의 요소들만 있게 된다. 탄창에서와 같이 대기렬은 많은 연산들이 진행됨에 따라 대기렬이 주기적으로 작아 진다.

간단한 해결방법은 언제나 front나 back가 배렬의 끝에 도달할 때마다 그것을 대기렬의 시작위치로 넘기는것이다(대기렬의 휘감기표시라고도 한다.). 다음의 표들은 어떤 연산들이 진행될 때의 대기렬의 상태를 보여 준다. 이것을 순환대기렬이라고 한다.

법을 리용한다. 실례로 어떤 프로그람작성자들은 대기렬의 크기를 기억하기 위한 입력점을 리용하지 않는데 그들은 대기렬이 비면 back = front − 1이라는 기초조건을 리용한다. 그 크기는 back와 front를 비교하는것으로써 암시적으로 계산된다. 이것은 아주 나쁜 방법인데 그것은 거기에 몇가지 특별한 경우가 있기때문이다. 만일 이 방법을 리용하여 코드를 수정하려면 대단히 주의하여야 한다. 만일 currentSize가 명시적인 자료성원이 아닐때 그 대기렬에 theArray.length ( ) − 1개의 요소들이 있으면 충만으로 되는데 그것은 theArray.length( )의 크기들이 서로 다르고 이것들가운데서 어떤것은 0이기때문이다. 프로그람작성자가 즐겨 쓰는 어떤 표현법을 정확히 선택하고 모든 루틴들이 포함되도록 서술한다. 이것을 실현하기 위한 몇가지 방법들이 있는데 만일 currentSize성원을 리용하지 않을 때에는 대체로 코드내에 한개 또는 두개의 주해를 주는것이 좋다.

enqueue연산들의 호출회수가 대기렬의 크기보다 더 크지 않는것이 확실한 응용들에서는 휘감기표식이 필요 없다. 탄창에서처럼 dequeue연산들은 호출루틴들이 대기렬이 비지 않았다는것이 명백하지 않으면 아주 묘하게 실행된다. 따라서 오유검사호출들은 중요한 코드를 제외하고 이 연산들을 주기적으로 뛰여 넘는다. 이것을 일반적으로 옳은것이라고 볼수 없는 리유는 이러한 오유검사가 진행된다고 하여도 사용자가 바라는대로 시간을 절약할수 없기때문이다.

여기에서 몇가지 대기렬루틴들을 서술하는것으로 이 절을 끝낸다. 다른 루틴들은 직결코드로 서술할수 있다. 우선 프로그람 3-41에 대기렬클라스의 대면부를 준다. 구축자들과 makeEmpty는 프로그람 3-42에 준다. back가 front의 앞에서 미리 1로 초기화되였다. 다음 연산은 enqueue루틴이다.

```
template <c1ass Object>
class Queue
{
   public:
         explicit Queue( int capacity = 10 );
         bool isEmpty( ) const;
         boot isFu11( ) const;
         const Object & getFront( ) const;

           void makeEmpty( );
           Object dequeue( );
           void enqueue( const Object & x );
      private:
           vector<Object> theArray;
           int                currentSize;
           int                front;
           int                back;
           void increment( int & x );
};
```

프로그람 3-41. 배렬로 실현한 대기렬의 클라스대면부

```
/**
 * Construct the queue.
 */
template <c1ass Object>
Queue<Object>::Queue( int capacity ) : theArray( capacity )
{
        makeEmpty( );
}

/**
 *   Make the queue logically empty,
 */
template <c1ass Object>
void Queue<Object>::makeEmpty( )
{
      currentSize = 0;
      front = 0;
      back = -1;
}
```

프로그램 3-42. 배렬로 실현한 구축자와 빈 대기렬만들기루틴

우에서 서술한 내용에 따라 프로그람 3-43에 enqueue루틴을 주었다. 마지막으로 dequeue루틴을 프로그람 3-44에 보여 준다.

```
/**
 * Insert x into the queue.
 * Throw Overflow if the queue is full.
 */
template <c1ass Object>

void Queue<Object>::enqueue( const Object & x )
{
          if( isFull( ) )
          throw Overf1ow( );
          increment( back );
          theArray[ back ] = x;
          currentSize++;
}
/**
 * Internal method to increment x with wraparound.
 */
template <c1ass Object>
void Queue<Object>: :increment( int & x )
{
```

```
            if( ++x == theArray.size( ) )
                  x = 0;
   }
```

프로그램 3-43. 배렬로 실현한 대기렬에 요소를 넣기 위한 루틴

```
/**
  * Return and remove the least recently inserted item from the queue .
  * Throw Underflow if the queue is empty.
  */
template <c1ass Object>
Object Queue<Object>::dequeue( )
{
              if( isEmpty( ) )
                     throw Underflow( );
              currentSize--;
              Object frontItem = theArray[ front ];
              Increment( front );
              return frontItem;
}
```

프로그램 3-44. 배렬로 실현한 대기렬에서 나오기루틴

3. 대기렬의 응용

효과적인 실행시간을 주기 위하여 대기렬을 리용한 알고리듬들은 많다. 그중 몇가지는 그라프리론에서 고찰할수 있는데 그것들은 제9장에서 설명한다. 여기에는 대기렬을 리용한 몇가지 간단한 실례들을 준다.

인쇄기에 일감들이 제기되면 그것들은 도착한 순서로 배치된다. 따라서 본질적으로 일감들은 대기렬에 배치되여 행인쇄기에 보내진다.[12]

본질적으로 모든 실생활에서의 렬은 하나의 대기렬과 같다. 실례로 차표판매소에서의 렬은 먼저 도착한것이 먼저 봉사되기때문에 대기렬이라고 말할수 있다.

또 다른 실례는 콤퓨터망과 관련된것이다. 디스크가 장비된 파일봉사기라고 하는 콤퓨터에 개인용콤퓨터를 위한 많은 망설정프로그람들이 있다. 다른 콤퓨터들에서 사용자들은 먼저 도착한것이 먼저 봉사된다는 조건으로 파일들을 호출할수 있다. 그러한 자료구조가 대기렬이다.

---

[12] 일감들이 줄어 들수 있기때문에 본질적이라고 말한다. 이것은 대기렬의 중간으로부터의 삭제와 같은것인데 이것은 엄밀한 정의에 대한 위반이다.

그이상의 실례들은 다음과 같은것들이다.

- 많은 사람들이 전화하려고 할 때 전화기들이 모두 통화중이면 일반적으로 대기렬에 배치된다.
- 자원이 제한된 큰 대학들에서 학생들은 실험설비들이 모두 리용되고 있다면 기다림목록에 수표하여야 한다. 실험을 먼저 시작하여 가장 오래 한 학생은 실험이 끝나면 먼저 나가고 가장 오래 기다린 학생이 그다음 사용자로 지명된다.

대중봉사론(Queueing theory)이라고 하는 수학분야는 얼마나 많은 사용자들이 한줄로 기다려야 하는가와 그 줄이 얼마나 오래 유지되는가 하는 문제들에 확률적인 계산을 주기 위하여 취급된다. 그 결과는 얼마나 많은 사람들이 자주 그 렬에 도착하는가, 사용자가 봉사를 받게 되면 그 사용자를 처리하는데 얼마나 오랜 시간이 걸리는가 하는데 관계된다. 이 두 파라메터들은 확률적으로 작성된 함수들에 의하여 주어 진다. 간단한 경우 그 결과는 분석적으로 계산될수 있다. 간단한 실례는 하나의 회선을 가지고 전화를 할 때이다. 만일 회선이 통화중이면 호출자들은 어떤 최대한계이상 기다림줄에 배치된다. 이 문제에서 중요한것은 사람들이 수화기를 빨리 놓는것이라는것을 보여 주는것이기때문에 전화업무에서 중요하다.

만일 *k*개의 회선이 있는 경우에 이 문제를 푸는것은 훨씬 더 어렵다. 분석적으로 해결하기 어려운 문제들은 흔히 모의적인 방법으로 해결한다. 이 경우에 대기렬을 리용하여 모의한다. 만일 *k*가 크면 이것을 효과적으로 수행하기 위하여 다른 자료구조들이 필요할수도 있다. 제6장에서 이러한 모의방법에 대하여 고찰한다. 그때 여러가지 *k*값들에 대하여 모의를 진행하고 적당한 기다림시간을 주는 최소 *k*값을 선택한다.

대기렬에 대한 추가적인 리용들은 많으며 탄창에서와 마찬가지로 그런 간단한 자료구조는 아주 중요하게 쓰일수 있다는것을 잘 보여 준다.

요약

이 장에서는 ADT에 대한 개념을 서술하고 가장 일반적인 세가지 추상자료형들에 대한 개념을 설명하였다. 여기에서 기본은 추상자료형들의 실현을 그것들의 함수와 분리하는것이다. 프로그람은 연산들이 무엇을 처리하는가를 알아야 하지만 실제로 그것을 어떻게 처리하는가는 알 필요가 없다.

목록, 탄창, 대기렬은 대체로 모든 콤퓨터과학에서 세개의 기초적인 자료구조로 되며 많은 실례들에서 그것들의 리용이 제공되였다. 특히 함수호출들을 기억하는데 탄창을 어떻게 리용하는가와 재귀가 실지 어떻게 실현되는가를 고찰하였다. 그것은 수속형언어를 가능하게 만들기때문일뿐아니라 재귀를 어떻게 실현하는가를 아는것이 그 리용에 대한 많은 모호한 문제들을 많이 줄이기때문에 이것을 리해하는것은 중요하다. 비록 재귀

가 강력하다고 해도 완전히 자유로운 연산은 아니며 재귀에 대한 람용은 프로그람을 파괴하는 결과를 초래할수 있다.

련습문제

3-1. 옹근수들이 오름순서로 보관된 련결목록들 *L*과 *P*가 주어 졌다. printLots(*L*, *P*)연산은 *P*에 의해서 지적된 위치에 있는 *L*의 요소를 출력한다. 실례로 만일 *P* =1, 3, 4, 6이면 *L*에 있는 첫번째, 세번째, 네번째, 여섯번째 요소들이 출력된다. 함수 printLots(*L*, *P*)를 작성하시오. 공개적인 목록연산들만 리용하시오. 이 함수의 실행시간은 얼마인가?

3-2. 자료가 없는 련결만을 리용하여 두개의 린접요소들을 교환하시오.

ㄱ. 단순목록에서

ㄴ. 2중목록에서

3-3. ListItr클라스(지적자에 기초한)에 대하여 그것이 구축될 때 설정되는 자료성원으로서 List에 대한 참조를 추가하시오. 그다음 insert에서 그것을 검사하는 List클라스루틴들을 수정하시오. 여기서 ListItr파라메터는 정확한 목록을 참조하고 있다.

3-4. remove함수가 하나의 List에 응용될 때 그것은 삭제된 매듭을 참조하는 어떤 ListItr를 무효로 한다. 그러한 반복자를 무효(Stale)반복자라고 한다. 무효반복자에 대한 어떤 연산이 반복자의 current자료성원이 NULL인것과 마찬가지로 동작한다는것을 담보하는 상수시간알고리듬을 작성하시오. 무효반복자들이 많다는데 주의하시오. 그 알고리듬을 실현하기 위하여 수정하여야 할 클라스부분을 설명하시오.

3-5. 어떤 련결목록의 부분을 다른 련결목록에 련결하려고 한다고 하자(잘라서 붙이는 연산과 같은). 이때 자르는 부분의 시작위치와 자르는 부분의 끝위치, 붙여야 할 위치를 나타내는 세개의 ListItr파라메터를 고려하여야 하며 여기서 모든 반복자들은 값을 가지고 있고 자르는 항목들의 수는 0이 아니라고 하자.

ㄱ. List클라스들의 동료가 아닌 잘라서 붙이는 함수를 작성하시오. 이 알고리듬의 실행시간은 얼마인가.

ㄴ. 잘라서 붙이는 함수를 List클라스안에 작성하시오. 이 알고리듬의 실행시간은 얼마인가?

3-6. 단순련결목록에서 retreat연산을 실현하시오(retreat는 반복자를 뒤로 한개 매듭만큼 이동한다.). 그 연산은 선형시간을 가진다는것에 주의하시오.

3-7. retreat함수를 제공하는 하나의 반복자(뒤로 가면서)를 가지고 2중련결목록을

실현하시오. 여기서 header에 대응하는 tail(목록의 끝을 지적하는)을 포함하시오. tail과 last매듭들에 대응하는 반복자들을 되돌리는 함수들을 작성하시오. 그리고 insertBefore와 insertAfter함수들을 작성하시오.

3-8. List클라스에 removeNext루틴을 추가하시오. removeNext는 ListItr파라메터로 주어 진 위치의 다음 항목을 삭제한다. 오유들이 어떻게 조종되는가.

3-9. 두개의 정렬된 목록 L_1과 L_2를 주고 기초적인 목록연산들만을 리용하여 $L_1 \cap L_2$을 계산하는 함수를 작성하시오.

3-10. 두개의 정렬된 목록 L_1과 L_2를 주고 기초적인 목록연산들만을 리용하여 $L_1 \cup L_2$를 계산하는 함수를 작성하시오.

3-11. 두개의 다항식들을 더하는 함수를 작성하시오. 입력은 변경시키지 말고 련결목록실현을 리용하시오. 만일 다항식들이 각각 M과 N개의 마디들로 되여 있으면 프로그람의 시간복잡도는 얼마인가?

3-12. 련결목록실현을 리용하여 두개의 다항식을 곱하는 함수를 작성하시오. 출력다항식이 지수적으로 정렬되고 같은 차수의 제곱항목은 많아서 하나가 되도록 하시오.

ㄱ. 이 문제를 $O(M^2N^2)$시간에 푸는 알고리듬을 주시오.

*ㄴ. $O(M^2N)$시간에 실행되는 곱하기프로그람을 작성하시오. 여기서 M은 두 다항식가운데서 작은 마디들을 가지는 다항식의 마디개수이다.

*ㄷ. $O(MN\log(MN))$시간에 곱하기를 실행하는 프로그람을 작성하시오.

ㄹ. 우에서 어느 시간한계가 가장 좋은가?

3-13. 다항식 $f(x)$를 가지고 $(f(x))^P$를 계산하는 프로그람을 작성하시오. 그 프로그람의 시간복잡도는 얼마인가. $f(x)$와 P를 적당히 선택하고 이 문제를 푸는 방안을 한개이상 제기하시오.

3-14. 임의의 정확도를 가지는 옹근수계산체계를 작성하시오. 다항식계산과 류사한 수법을 리용할수 있다. 2^{4000}에서 0부터 9까지의 수자분포상태를 계산하시오.

3-15. **죠세프(*josephus*)의 문제**는 다음과 같은 유희문제이다. 1부터 N까지 번호가 붙은 N명의 사람이 원주안에 있다. 1번부터 시작해서 손수건을 넘겨 준다. M번 넘기기를 진행한 다음 그 손수건을 쥐고 있는 사람을 그 원주에서 내보내고 다시 원을 만든 다음에 내보낸 사람의 다음 사람부터 다시 경기를 시작한다. 마지막에 남는 사람이 경기의 승리자이다. 이와 같이 경기를 진행하면 M=0, N=5인 때 경기자들은 번호순서대로 원주에서 나가게 되며 따라서 마지막 다섯번째의 선수가 승리자로 된다. 만일 M=1이고 N=5이면 경기자들은 2, 4, 1, 5의 순서로 나가게 된다.

ㄱ. 일반적인 값 M과 N을 가지고 조세프의 문제를 푸는 프로그람을 작성하시오. 가능한껏 효과가 큰 프로그람을 만들기 위해 노력하시오. 그리고 요소들을 정확히 처리하시오.

ㄴ. 그 프로그람의 실행시간은 얼마인가?

ㄷ. 만일 M=1이면 프로그람의 실행시간은 얼마인가. 큰 값 N(N>1000)에 대하여 delete루틴은 실제속도에 어떤 영향을 미치는가?

3-16. 단순련결목록에서 개별적인 요소를 탐색하는 프로그람을 작성하시오. 재귀적인 방법과 비재귀적인 방법으로 이것을 처리하고 실행시간을 비교하시오.

3-17. ㄱ. $O(N)$시간안에 단순련결목록의 지적자들의 방향을 전도시키는 비재귀적인 함수를 작성하시오.

*ㄴ. 상수적인 림시공간을 리용하여 $O(N)$시간안에 단순련결목록을 전도시키는 함수를 작성하시오.

3-18. 사회안전번호에 의해 학생기록들에 대한 배렬을 정렬한다고 하자. 1,000개의 바께쯔들과 세 단계를 가지는 밑수정렬을 리용하여 이것을 처리하는 프로그람을 작성하시오.

3-19. ㄱ. 자동조종식목록에 대한 배렬실현을 작성하시오. 자동조종식목록은 삽입이 모두 앞에서만 실행되는것을 제외하고는 보통의 목록과 류사하며 find에 의해서 요소가 호출될 때 그것은 다른 항목들에 대하여 상대적인 순서를 수정함이 없이 목록의 앞에서 삭제된다.

ㄴ. 자동조종식목록에 대한 련결목록실현을 작성하시오.

*ㄷ. 매개 요소가 호출될 때 고정된 확률 P_i를 가진다고 하자. 가장 큰 호출확률을 가진 요소들은 앞으로 다가간다는것을 고찰하시오.

3-20. 삭제에 대한 또 하나의 방법은 **지연삭제**(lazy deletion)를 리용하는것이다. 어떤 요소를 삭제하기 위하여 림시적인 비트마당을 리용하여 삭제될 요소를 표시한다. 목록에서 삭제된 요소들의 수와 삭제되지 않은 요소들의 수는 그 자료구조의 부분으로 유지된다. 만일 삭제된 요소들이 삭제되지 않은 요소들만큼 많으면 모든 표식 붙은 매듭들에 대하여 표준적인 삭제알고리듬을 실행하기 위하여 전체 목록을 순회한다.

ㄱ. 지연삭제의 우점과 결함을 지적하시오.

ㄴ. 지연삭제를 리용하여 표준적인 련결목록연산들을 실현하기 위한 루틴들을 작성하시오.

3-21. 다음의 언어로 기호들의 균형잡기를 검사하는 프로그람을 작성하시오.

ㄱ. Pascal(begin/end, ( ), [ ], { })

ㄴ. C++(/* */, ( ), [ ], { })

*ㄷ. 대략적인 근거를 반영하는 오유통보문을 출력하는 방법을 설명하시오.

3-22. 뒤배치식을 계산하는 프로그람을 작성하시오.

3-23. ㄱ. (,), +, -, *, /가 포함된 사이배치식을 뒤배치식으로 변환하는 프로그람을 작성하시오.

ㄴ. 연산범위에 지수연산자를 추가하시오.

ㄷ. 뒤배치식을 사이배치식으로 변환하는 프로그람을 작성하시오.

3-24. 한개의 배렬만을 가지고 두개의 탄창을 실현하는 루틴들을 작성하시오. 탄창루틴들은 배렬에 있는 매개 요소들이 다 리용되지 않으면 자리넘침을 선언하지 않아도 된다.

3-25. *ㄱ. push와 pop, 그리고 자료구조안에서 가장 작은 요소를 되돌리는 findMin 연산들이 모두 최악의 경우에 $O(1)$시간을 가지도록 하는 자료구조를 제기하시오.

*ㄴ. 가장 작은 요소를 찾아서 삭제하는 네번째 연산 deleteMin이 추가되면 그 연산들중의 하나가 적어도 $\Omega(\log N)$의 시간을 가진다는것을 증명하시오(이것은 제7장부분을 고찰할것을 요구한다.).

***3-26.** 하나의 배렬에서 세개의 탄창을 실현하는 방법을 고찰하시오.

3-27. 제2장 제4절에서 $N = 50$을 가지고 피보나치수들을 계산하기 위한 재귀루틴은 탄창공간이 부족하게 되겠는가? 그 리유는 무엇인가?

3-28. **쌍방향대기렬**(deque)에서는 다음의 연산들을 포함하게 된다.

push(x): 2중대기렬의 앞끝에 항목 x를 삽입

pop(): 2중대기렬로부터 첫번째 요소를 삭제하고 그것을 되돌리기

inject(x): 2중대기렬의 꼬리에 요소 x를 삽입

eject(): 2중대기렬로부터 꼬리요소를 삭제하고 그것을 되돌리기

매 연산당 $O(1)$의 시간을 가지는 2중대기렬을 제공하는 루틴들을 서술하시오.

3-29. 배렬에 기초한 탄창은 배렬의 용량이 배렬한계에 도달하면 그 한계를 벗어난다. 다음의 방안을 생각해 보시오. resize산법을 리용하여 더 큰 배렬을 만드시오. 더 큰 배렬을 만드는 resize의 값은 새로운 크기를 되돌린다.

ㄱ. 배렬의 크기를 한개 요소만큼 확장하는 경우 N개의 요소를 삽입할 때 최악의 경우 실행시간은 얼마인가?

ㄴ. 배렬의 크기가 5개 요소만큼 확장되는 경우 N개 요소를 삽입할 때의 최악의 경우 실행시간은 얼마인가?

ㄷ. 배렬의 크기가 2배(그 용량이 령이 아닌 때)로 되여 N개 요소를

삽입할 때 최악의 경우의 실행시간은 얼마인가?

ㄹ. 우의 방법들가운데서 가장 효과적인 방안을 실현하시오.

*3-30. 대기렬에 대한 련습문제 3-29를 수정하시오. resize산법을 실행한 다음 요소들을 이동할 필요가 있다는데 대하여 주의하시오.

3-31. List를 자료구조로 하여 효과적인 탄창클라스를 실현하시오.

3-32. 선두매듭을 가지지 않는 련결목록을 리용하여 대기렬을 실현하시오. 대기렬의 앞과 뒤의 ListNode들에 대한 지적자들을 보관하시오. 빈 대기렬에 대한 특별한 경우의 조작에 주의하시오.

3-33. 자료성원으로써 List와 그 List에서 마지막위치를 나타내는 ListItr를 리용하여 효과적인 대기렬클라스를 실현하시오.

3-34. 련결목록은 어떤 매듭 P로부터 시작하여 매듭 P로 되돌아 가는 next련결들에 따르는 순환을 포함한다. P는 목록에서 첫번째 매듭을 가지지 말아야 한다. N개 매듭을 포함하는 련결목록이 주어 졌다고 하자. 그러나 N의 값은 알려지지 않았다.

ㄱ. 목록이 순환을 포함하는가를 결정하는 $O(N)$알고리듬을 작성하시오.

*ㄴ. 다만 $O(1)$의 림시공간만을 리용하여 ㄱ를 다시 작성하시오(암시: 정지속도가 각이하기때문에 목록의 시작에서 2개의 반복자를 리용하시오.).

3-35. 대기렬을 실현하는 한가지 방법은 순환련결목록을 리용하는것이다. 목록이 선두매듭을 가지지 않으며 목록에서 매듭에 대응하는 반복자가 많아서 한개라고 하자. 다음의 표현들에서 기본적인 대기렬연산들을 최악의 경우 상수적인 시간에 모두 수행할수 있는가? 명백한 대답을 주시오.

ㄱ. 목록의 첫번째 요소에 대응하는 반복자를 유지하시오.

ㄴ. 목록의 마지막요소에 대응하는 반복자를 유지하시오.

3-36. 단순련결목록에서 마지막매듭이 아니라는것을 담보하는 어떤 매듭에 대한 지적자가 있다고 하자. 이때 어떤 다른 매듭들에 대한 지적자는 가지지 않는다(련결들에 따르는것은 제외하고). 련결목록의 완전한 상태를 유지하면서 그 련결목록으로부터 그 매듭에 보관된 값을 론리적으로 삭제하는 $O(1)$알고리듬을 작성하시오.

3-37. 단순련결목록이 선두매듭과 꼬리매듭을 가지고 실현된다고 하자. 다음의 상수시간알고리듬을 작성하시오.

ㄱ. 반복자로 주어 진 위치 p앞에 요소 x를 삽입하시오.

ㄴ. 위치 p(반복자로 주어 진)에 보관된 항목을 삭제하시오.

제4장. 나무

입력의 개수가 많은 경우 련결목록에 대한 선형적인 접근시간은 아주 크다.

이 장에서는 대부분 연산들의 실행시간이 평균 $O(\log N)$인 간단한 자료구조를 본다. 또한 최악의 경우에도 우와 같은 시간한계를 담보하는 이 자료구조의 변형에 대하여 개념적으로 간단히 고찰하고 명령들의 긴 서렬에 대하여 기본상 매 연산이 $O(\log N)$의 실행시간을 가지는 두번째 변형을 고찰한다.

여기서 언급하는 자료구조를 **2진탐색나무**(*Tree Binary Search*)라고 한다. 일반적으로 나무는 콤퓨터과학에서 매우 쓸모 있는 추상화이므로 다른 보다 일반적인 응용들을 고찰한다. 이 장에서는 다음과 같은 문제들을 설명한다.

- 몇가지 일반조작체계에서 나무를 리용한 파일체계의 실현
- 수학식계산에서 나무의 리용
- 평균 $O(\log N)$시간에 탐색연산을 진행하는데서 나무를 리용하는 방법과 최악의 경우에도 $O(\log N)$한계를 얻을수 있도록 이 사고방식을 개선하는 문제, 또한 자료가 외부기억에 보관될 때 이런 연산들의 실현방법

제1절. 예비적인 준비

나무는 여러가지 방법으로 정의할수 있다. 나무를 정의하는 제일 자연적인 방법은 재귀적인 방법이다. 나무는 매듭들의 모임이다. 그 모임은 비여 있거나 **뿌리매듭**(*root*)이라고 하는 특수한 매듭 r와 령 또는 그이상의 비지 않은 부분나무 T_1, T_2, …, T_k 들을 가지고 있다. 부분나무의 매개 뿌리매듭들은 r로부터 방향변에 의하여 련결된다.

매 부분나무의 뿌리매듭을 r의 자식이라고 하며 r는 매 부분나무의 뿌리매듭의 부모라고 한다. 그림 4-1은 재귀적인 정의를 리용한 전형적인 나무를 보여 준다.

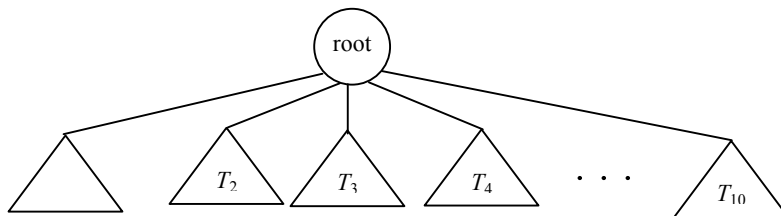



그림 4-1. 일반나무

재귀정의로부터 나무는 N개의 매듭(그중 하나는 뿌리매듭이다.)들과 N-1개의 변들의 모임이라는것을 알수 있다. N-1개의 변들이 있다는것은 그 매개 변이 매듭들을 그의 부모와 련결시키며 뿌리매듭을 제외한 매 매듭은 하나의 부모를 가진다는 사실을 동반한다(그림 4-2를 보시오.).

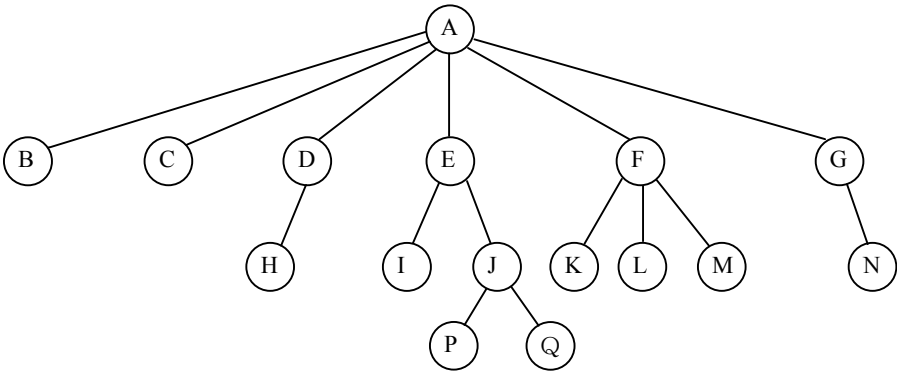



그림 4-2. 나무

그림 4-2의 나무에서 뿌리매듭은 A이다. 매듭 F는 부모로서 A를, 자식으로서 K, L, M을 가진다. 매 매듭은 0을 포함하여 임의의 개수의 자식을 가진다. 자식이 없는 매듭들을 잎매듭이라고 하는데 그림 4-2의 나무에서 잎매듭들은 $B, C, H, I, P, Q, K, L, M, N$이다. 같은 부모를 가지는 매듭들을 형제라고 하는데 K, L, M은 모두 형제이다. 조부모와 손자들도 이와 류사한 방법으로 정의할수 있다.

매듭 n_1에서 n_k까지의 경로는 $1 \leqq i < k$일 때 n_i가 n_{i+1}의 부모로 되는 매듭들의 서렬 $n_1, n_2, \cdots, n_k$로 정의된다. 이 경로의 길이는 경로상에 있는 변들의 개수 즉 k-1이다. 매개 매듭으로부터 자기자신까지의 경로의 길이는 0이다. 나무에서 뿌리매듭으로부터 매개 매듭까지의 경로는 오직 하나뿐이라는것을 명심하시오.

어떤 매듭 n_i에 대하여 그의 깊이는 뿌리매듭으로부터 n_i까지 유일한 경로의 길이이다. 따라서 뿌리매듭의 깊이는 0이다. n_i의 높이는 n_i로부터 잎매듭까지의 가장 긴 경로의 길이이다. 따라서 모든 잎매듭들은 높이가 0이다. 나무의 높이는 뿌리매듭의 높이와 같다. 그림 4-2의 나무에서 E는 깊이가 1이고 높이가 2이며 F는 깊이가 1이고 높이가 1이며 나무의 높이는 3이다. 나무의 깊이는 가장 깊은 잎매듭의 깊이와 같은데 그 값은 항상 나무의 높이와 같다.

만일 n_1부터 n_2까지 경로가 있으면 n_1은 n_2의 선조이고 n_2는 n_1의 자손이다. $n_1 \neq n_2$이면 n_1은 n_2의 고유한 선조이고 n_2는 n_1의 고유한 자손이다.

1. 나무의 실현

나무를 실현하는 한가지 방법은 매개 매듭안에 그의 자료외에 그 매듭의 매개 자식들에 대한 련결을 가지는것이다. 그러나 매개 매듭당 자식들의 수가 크게 변하고 그 수를 미리 알수 없으므로 자료구조안에서 자식들을 직접 련결할수 없다. 그것은 많은 기억공간이 랑비되기때문이다. 간단한 해결책은 매 매듭의 자식들을 나무매듭들의 련결목록에 보존하는것이다. 프로그람 4-1에 그에 대한 전형적인 선언을 주었다.

```
Struct TreeNode
{
            Object       element;
            TreeNode *firstChild;
            TreeNode *NextSibling;
};
```

프로그람 4-1. 나무에 대한 매듭선언

그림 4-3은 나무가 이러한 실현에서 어떻게 표현되는가를 보여 준다. 아래쪽으로 향하는 화살들은 firstChild련결들이고 왼쪽에서 오른쪽으로 향하는 화살들은 nextSibling련결들이다. Null련결들은 너무 많으므로 표시하지 않는다.

그림 4-3의 나무에서 매듭 *E*는 형제(*F*)에 대한 련결과 자식(*I*)에 대한 련결을 둘다 가지며 어떤 매듭들은 아무것도 가지지 않는다.

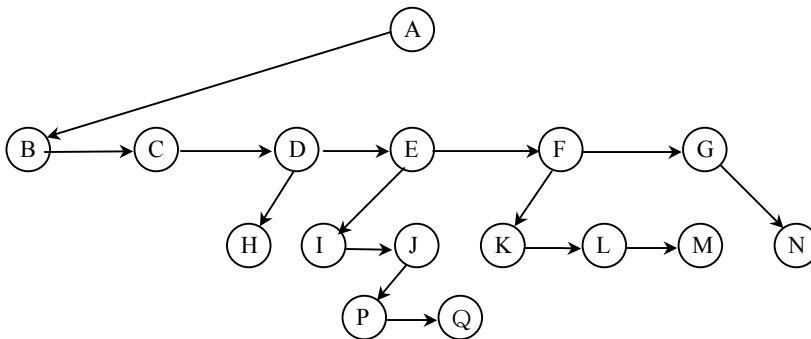



그림 4-3. 그림 4-2에서 보여 준 나무에 대한 첫번째 자식과 다음번 형제의 표현

2. 나무의 순회와 응용

나무에 대하여서는 많은 응용들이 있다. 그 일반적인 리용의 하나는 UNIX와

VAX/VMS, DOS를 비롯하여 많은 조작체계들에서의 등록부구조이다. 그림 4-4는 UNIX 파일체계에서 리용하는 전형적인 등록부이다.

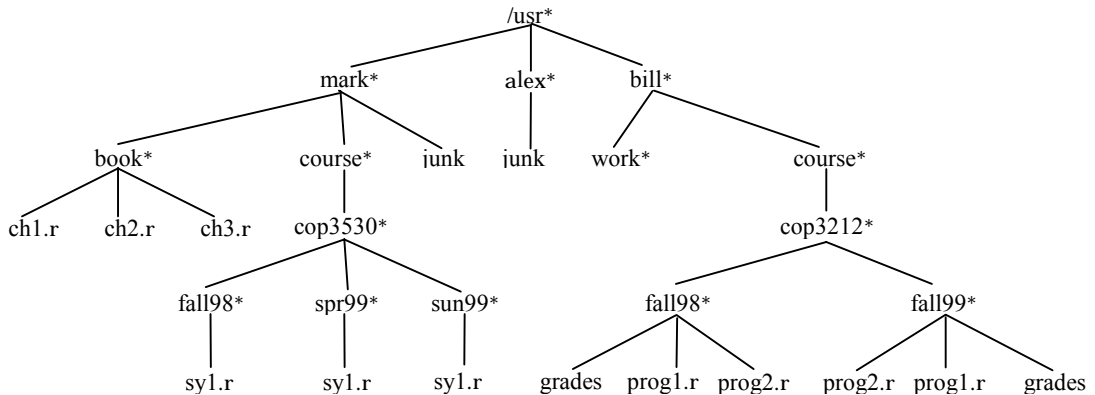



그림 4-4. UNIX등록부

이 등록부에서 뿌리는 */usr*이다(이름옆의 별표는 */usr*가 등록부 그자체라는것을 가리킨다.). */usr*는 세개의 자식들 *mark, alex, bill*을 가지는데 그것들자체도 등록부이다. 따라서 */usr*는 세개의 등록부를 포함하며 파일은 없다. 파일이름 */usr/mark/book/ch1.r*는 가장 왼쪽에 있는 자식들을 따라 세번 내려 가면 얻어 진다. 첫번째 이름다음에 매개 /는 변을 의미하며 그 결과는 완전한 경로이름이다. 이러한 계층적인 파일체계는 사용자가 자기 자료를 론리적으로 조직하게 하므로 매우 통속적이다. 더우기 각이한 등록부에 있는 두개의 파일들은 그것들이 뿌리로부터 서로 다른 경로를 가지며 따라서 경로이름도 다르기때문에 같은 이름을 가질수 있다. UNIX파일체계에서 등록부는 자기의 모든 자식들에 대한 목록을 가지는 어떤 파일이다. 그러므로 등록부들은 우의 형선언과 거의 정확히 일치하게 구조화된다.[13] 실지 UNIX의 일부 판본에서는 파일을 인쇄하는 표준지령이 등록부에 적용되면 그 등록부안의 파일이름들이 출력상에 나타날수 있다(다른 비ASCII정보를 가지고).

이제 등록부에 있는 모든 파일들의 이름을 표시하려고 한다고 하자. 출력형식은 깊이 d_i를 가지는 파일들은 그 이름을 d_i만큼 들여 쓰는것이다. 그 알고리듬을 가상코드로 써 프로그람 4-2에 보여 주었다.

재귀함수 listAll은 뿌리매듭에 대해서는 들여 쓰기를 하지 않도록 깊이 0에서 시작하여야 한다. 이 깊이는 내부처리용변수이며 호출루틴이 알려고 하는 파라메터는 아니다.

[13] UNIX파일체계에서 매개 등록부는 또한 자기자체를 가리키는 하나의 입력점과 그 등록부의 부모를 가리키는 다른 하나의 입력점을 가진다. 따라서 학술적으로 볼 때 UNIX파일체계는 나무가 아니지만 나무처럼 고찰한다.

따라서 depth에 대하여 암시적인 값 0이 주어 진다.

```
              void FileSystem::listA11( int depth = 0 ) const
              {
/*1*/                printName( depth ); // Print the name of the object
/*2*/                if( isDirectory( ) )
/*3*/                for each file c in this directory (for each child)
/*4*/                     c.listAll( depth + 1 );
              }
```

프로그람 4-2. 계층적인 파일체계에서 등록부를 표시하는 가상코드

알고리듬의 론리는 다음과 같다. 파일객체의 이름은 적당한 수의 꼬리표와 함께 인쇄된다. 만일 입력점이 등록부이면 모든 자식들을 재귀적으로 하나씩 처리한다. 이 자식들은 한준위 더 깊으므로 한 여분공간만큼 들여 쓰기된다. 그 출력결과를 그림 4-5에 보여 준다.

```
/usr
     mark
          book
               ch1.r
               ch2.r
               ch3.r
          course
               cop3530
                    fall98
                         sy1.r
                    spr99
                         sy1.r
                    sum99
                         sy1.r
          junk
     alex
          junk
     bill
          work
          course
               cop3212
                    fall98
                         grades
                         prog1.r
                         prog2.r
                    fall99
                         prog2,r
                         prog1.r
                         grades
```

그림 4-5. 등록부목록(선뿌리순회)

이 순회방법을 **선뿌리순회**(*preorder*)라고 한다. 선뿌리순회에서 매듭에 대한 처리는 그 자식들이 처리되기전에 수행된다. 이 프로그람이 실행될 때 1행은 매 매듭당 정확히 한번만 실행되므로 매개 이름은 한번만 출력된다. 1행이 매 매듭당 기껏해서 한번 실행되기때문에 2행도 역시 매 매듭당 1번 실행된다. 더우기 4행은 매개 매듭의 매 자식에 대하여 기껏해서 한번 실행된다. 그러나 자식들의 수는 매듭들의 수보다 정확히 하나만큼 작다. 마지막으로 for순환고리는 4행의 실행 하나만을 반복하며 한 순환이 끝날 때마다 하나씩 증가한다. 따라서 전체 처리량은 매개 매듭에 대하여 상수적이다. 만일 출력할 파일이름이 N개라면 실행시간은 $O(N)$으로 된다.

나무를 순회하는 또 다른 일반적인 방법은 **후뿌리순회**(*postorder*)이다. 후뿌리순회에서 매듭에 대한 처리는 그 자식들이 처리된 다음에 수행된다. 실례로 그림 4-6은 앞에서와 같은 등록부구조를 표

현하는데 괄호안의 수자들은 매개 파일에 의해서 리용된 디스크블로크들의 수이다.

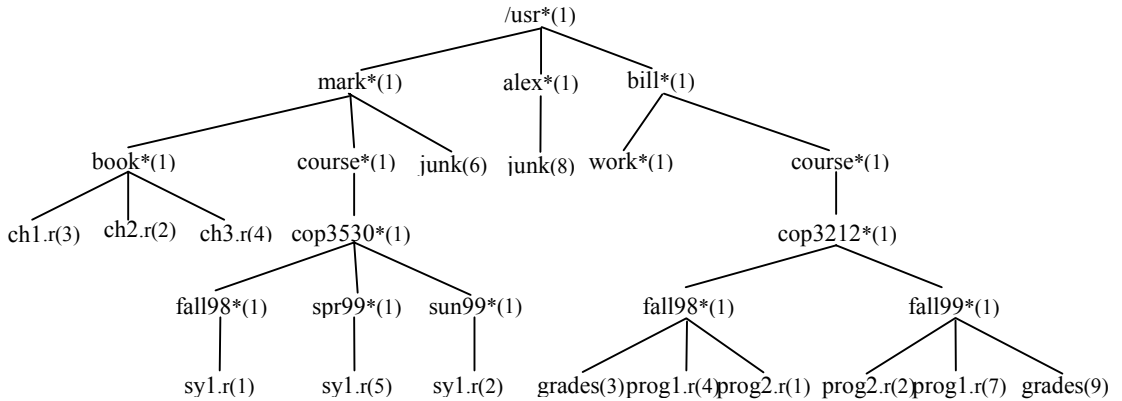



그림 4-6. 후뿌리순회로 주어 진 파일크기를 가지는 UNIX등록부

등록부 그자체가 파일이므로 그것들역시 크기를 가진다. 나무에서 모든 파일들에 의해서 리용된 총 블로크들의 수를 계산한다고 하자. 이를 위한 가장 보편적인 방법은 보조등록부들 */usr/make*(30)와 */usr/alex*(9), */usr/bill*(32)에 포함되는 블로크수를 알아 내는것이다. 그러면 총 블로크수는 전체 보조등록부들 71에 */usr*에 리용된 하나의 블로크를 더하여 총계 72로 된다. 프로그람 4-3에서 가상코드산법 size가 이 방법을 실현한다.

```
        int FileSystem::size( ) const
        {
/*1*/       int totalSize = sizeOfThisFi1e( );
/*2*/       if( isDirectory( ) )
/*3*/           for each file c in this directory (for each child)
/*4*/           totalSize += c.size( );

/*5*/       return totalSize;
        }
```

프로그람 4-3. 등록부의 크기를 계산하기 위한 가상코드

만일 현재의 객체가 등록부가 아니면 size는 단순히 현재의 객체에서 리용하는 블로크수를 되돌린다. 한편 등록부에 의해서 리용된 블로크수는 모든 자식들에서 재귀적으로 찾아 낸 블로크수에 더해 진다. 후뿌리순회방법과 선뿌리순회방법의 차이를 고찰하기 위하여 그림 4-7에서는 매 등록부나 파일의 크기가 알고리듬에서 어떻게 계산되는가를 보여 준다.

```
            ch1.r                    3
            ch2.r                    2
            ch3.r                    4
        book                        10
                    sy1.r            1
                fall98               2
                    sy1.r            5
                spr99                6
                    sy1.r            2
                sum99                3
            cop3530                 12
        course                      13
         junk                        6
    mark                            30
         junk                        8
     alex                            9
         work                        1
                    grades           3
                    prog1.r          4
                    prog2.r          1
                fall98               9
                    prog2.r          2
                    prog1.r          7
                    grades           9
               fall99               19
            cop3212                 29
        course                      30
    bill                            32
/usr                                72
```

그림 4-7. size 함수의 결과

제2절. 2진나무

2진나무는 어느 매듭도 자식을 둘이상 가질수 없는 나무이다.

그림 4-8은 1개의 뿌리와 2개의 부분나무로 이루어 진 2진나무를 보여 주는데 T_L과 T_R는 빈나무일수도 있다.

2진나무의 중요한 성질은 균형2진나무의 깊이가 N보다 상당히 작은것이다. 분석해 보면 그 평균깊이가 $O(\sqrt{N})$이고 **2진탐색나무**(*binary search tree*)라고 하는 특수한 형태의 2진나무에서는 평균깊이가 $O(\log N)$이라는것을 보여 준다. 그러나 그 깊이는 그림 4-9의 실례에서처럼 N-1만큼 커질수도 있다.

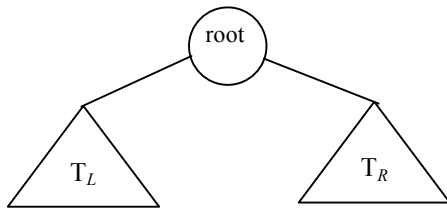



그림 4-8. 일반적인 2진나무

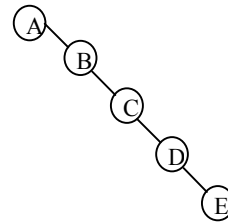



그림 4-9. 최악의 경우의 2진나무

1. 2진나무의 실현

2진나무가 많아서 두개의 자식을 가지기때문에 그 자식들에 대한 직접련결을 가질 수 있다. 나무매듭들의 선언은 2중련결목록구조와 류사하며 매 매듭의 구조는 element정보와 다른 매듭들에 대한 두개의 지적자(left와 right)로 이루어 진다(프로그람 4-4를 보시오.).

```
struct BinaryNode
{
    Object          element;     // The data in the node
    BinaryNode *1eft;           // Left child
    BinaryNode *right;          // Right child
};
```

프로그람 4-4. 2진나무매듭클라스(가상코드)

2진나무를 그릴 때 련결목록에서 관례적인 4각형도형을 리용할수 있겠지만 나무들이 실제로는 그라프이므로 보통 선으로 련결된 원도형으로 그린다. 또한 나무를 그릴 때 N개의 매듭을 가지는 2진나무가 N+1개의 NULL련결을 요구하므로 NULL련결은 표시하지 않는다.

2진나무들은 탐색과 관련 없는 많은 응용들에서 중요하게 리용된다. 2진나무들의 주요한 리용의 하나가 번역기설계분야인데 다음 부분에서 보게 된다.

2. 실례: 수식나무

그림 4-10은 **수식나무**(expression tree)의 실례이다. 수식나무에서 잎매듭들은 상수 또는 변수이름과 같은 피연산수들이며 다른 매듭들은 연산자들을 포함한다. 이러한 실천적인 나무는 모든 연산자들이 두개의 항으로 이루어 지기때문에 2진나무표현이 가능하며 이러한 표현방법은 두개이상의 자식들을 가지는 매듭들에서도 가능하다. 또한 부정연산자(unary minus operator)와 같이 하나의 자식만을 가지는 매듭에 대해서도 가능하다. 뿌리

매듭의 연산자를 그의 왼쪽과 오른쪽의 부분나무들을 재귀적으로 평가하여 얻은 값들에 적용하면 수식나무 *T*를 평가할수 있다. 이 실레에서 왼쪽 부분나무는 a + (b * c)로 평가되고 오른쪽 부분나무는 ((d * e) + f) * g로 평가된다. 따라서 전체 나무는 (a + (b * c)) + (((d * e) + f) * g)로 표현된다.

여기서 괄호안에 든 왼쪽 식을 재귀적으로 만들어 내고 그다음 뿌리에 있는 연산자를 출력하며 마지막으로 괄호안에 든 오른쪽 식을 재귀적으로 만들어 냄으로써 사이배치(infix)표현을 만들어 낼수 있다. 이러한 방법(왼쪽, 매듭, 오른쪽)을 **중뿌리순회**(inorder)라고 하는데 그것이 만들어 내는 식형태로 하여 기억하기 쉽다.

또 다른 순회방법은 왼쪽 부분나무와 오른쪽 부분나무를 재귀적으로 출력하고 그다음에 뿌리매듭의 연산자를 출력하는것이다. 이 방법을 적용하면 출력은 a b c * + d e * f + g * +인데 이것은 제3장 제3절 3에서 서술한 뒤배치(postfix)표현이라는것을 쉽게 알수 있다. 이러한 순회방법을 일반적으로 **후뿌리순회**(postorder)라고 한다. 제4장 제1절에서 이미 이 순회방법을 보았다.

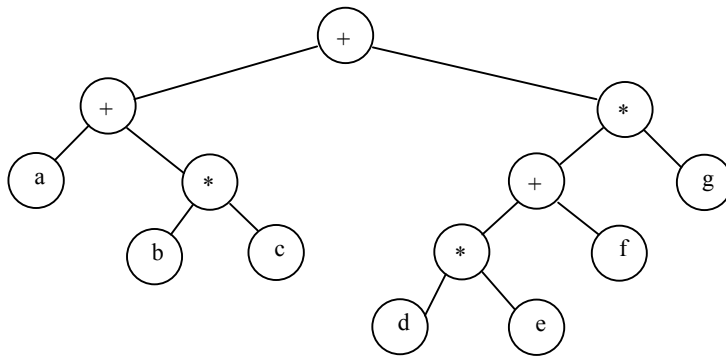



그림 4-10. (a + b * c) + ((d * e + f) * g)에 대한 표현나무

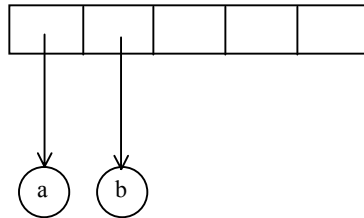
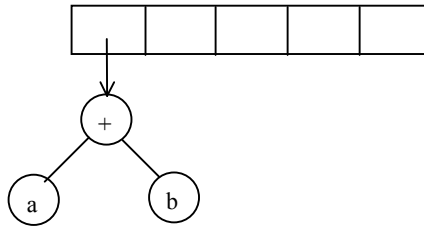
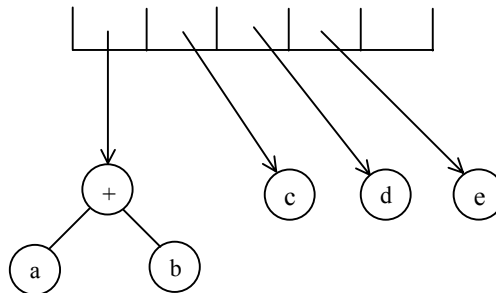
세번째 순회방법은 먼저 뿌리의 연산자를 출력하고 그다음에 왼쪽과 오른쪽 부분나무들을 재귀적으로 출력하는것이다. 표시결과는 + + a * b c * + * d e f g로서 이것은 보다 쓸모가 적은 앞배치표현(prefix form)이며 이러한 방법을 선뿌리순회(preorder)라고 한다. 이 방법도 역시 제4장 제1절에서 이미 고찰하였다. 이 장의 뒤부분에서 이 순회방법들을 다시 고찰한다.

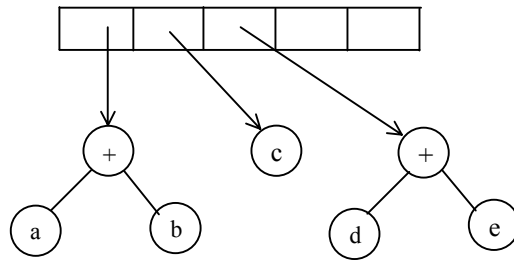
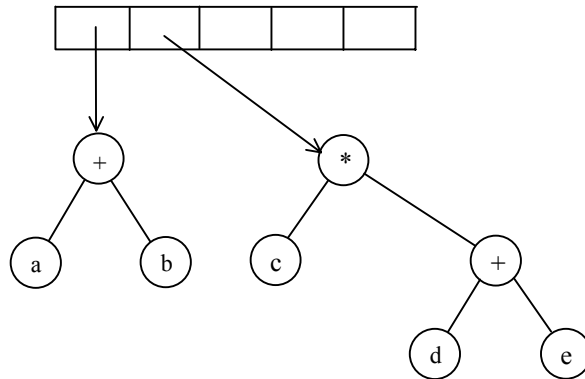
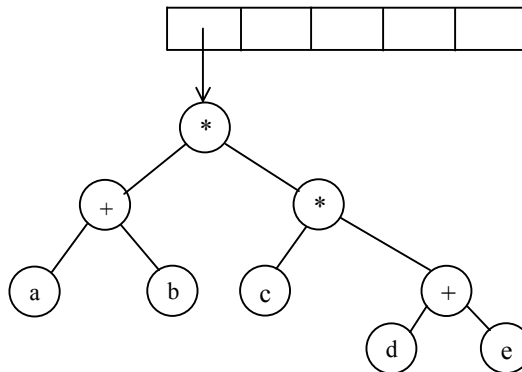
수식나무의 구축

여기서는 뒤배치표현을 수식나무로 변환하는 알고리듬을 준다. 사이배치표현을 뒤배치표현으로 변환하는 알고리듬은 이미 고찰하였으므로 두개의 일반적인 입력형식으로부터 수식나무를 만들어 낼수 있다. 이제 고찰하게 되는 산법은 제3장 제2절 3의 뒤배치표

현평가알고리듬과 대단히 류사하다. 한번에 하나의 기호를 읽어 들이면서 만일 그 기호가 피연산수이면 한매듭나무를 만들고 그에 대한 지적자를 탄창에 넣는다. 만일 기호가 연산자이면 탄창에서 두개의 나무 T_1과 T_2에 대한 지적자를 뽑아 내여(T_1이 먼저 꺼내 진다.) 뿌리가 연산자이고 왼쪽과 오른쪽 자식들이 T_2와 T_1을 가리키는 새로운 나무를 재귀적으로 형성하고 그 새로운 나무에 대한 지적자를 탄창에 넣는다.

실례로 입력이 a b + c d e + * *라고 하자.

첫 2개의 기호들은 피연산수들이므로 한개의 매듭으로 된 나무를 만들어 그에 대한 지적자를 탄창에 넣는다.[14]

다음 +가 읽어 지므로 나무들에 대한 2개의 지적자를 꺼내고 새로운 나무가 형성되며 그에 대한 지적자가 탄창에 넣어 진다.

다음 c와 d, e가 읽어 지고 한개 매듭으로 이루어 진 나무가 각각 만들어 져 그 나무들에 대한 지적자가 탄창에 넣어 진다.

다음에는 +가 읽어 지므로 2개의 나무들이 병합된다.

[14] 편리상 그림에서는 탄창이 왼쪽에서 오른쪽으로 증가하도록 한다.

계속하여 *가 읽어 지고 2개 나무의 지적자들을 뽑고 뿌리가 *인 새로운 나무를 형성한다.

나중에 마지막기호가 읽어 지고 2개 나무들이 병합되여 탄창에는 최종나무에 대한 지적자만이 남게 된다.

제3절. 탐색나무ADT-2진탐색나무

2진나무의 주요한 리용은 탐색이다. 나무에 있는 매개 매듭이 어떤 항목을 보존한다고 하자. C++에서는 그 어떤 복잡한 항목들도 쉽게 조종되지만 이 실례에서는 간단히 그 항목들이 옹근수들이라고 가정하자. 또한 모든 항목들은 따로따로 구별된다고 가정하

며 중복에 대해서는 후에 고찰한다.

2진나무를 2진탐색나무로 만드는 속성은 나무에 있는 매 매듭 *X*에 대하여 그의 왼쪽 부분나무에 있는 모든 항목들의 값이 *X*의 항목값보다 더 작고 오른쪽 부분나무에 있는 모든 항목들의 값은 *X*의 항목값보다 더 크게 하는것이다. 이것은 나무의 모든 요소들이 어떤 지정된 방식으로 순서화될수 있다는것을 의미한다. 그림 4-11에서 왼쪽에 있는 나무는 2진탐색나무이지만 오른쪽에 있는 나무는 그렇지 않다. 오른쪽에 있는 나무는 항목값이 6인 매듭의 왼쪽 부분나무에 요소값이 7인 매듭이 존재한다(이것은 뿌리매듭에서 발생한다.).

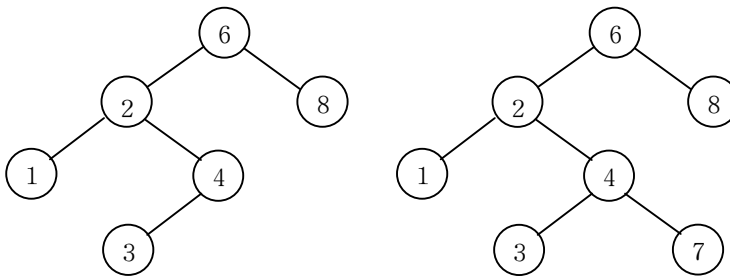



그림 4-11. 2개의 2진나무(왼쪽 나무만 탐색나무이다.)

여기서는 2진탐색나무들에서 흔히 쓰이는 연산들에 대하여 간단히 설명한다. 나무들에 대한 재귀적인 정의로 하여 이 루틴들도 재귀적으로 서술하는것이 보통이라는데 주목하시오. 2진탐색나무의 평균깊이가 $O(\log N)$이므로 탄창공간밖에서의 실행을 걱정할 필요는 없다.

프로그람 4-5는 BinaryNode클라스형판을ㄱ 보여 준다. 련결목록클라스에서와 마찬가지로 불완전한 클라스와 2진탐색나무클라스가 BinaryNode의 비공개자료성원에 접근할수 있는 권한을 주는 friend선언을 리용한다.

```
template <c1ass Comparab1e>
class BinarySearchTree;

template <c1ass Comparab1e>
class BinaryNode
{
        Comparable    element;
        BinaryNode *1eft;
        BinaryNode *right;
        BinaryNode( const Comparable & theElement, BinaryNode *lt,
                                                        BinaryNode *rt )
```

```
        : e1ement( theElement ), 1eft( 1t ), right( rt ) { };
      friend class BinarySearchTree<Comparab1e>;
    };
```

프로그람 4-5. BinaryNode 클라스

프로그람 4-6은 BinarySearchTree클라스형판의 대면부를 보여 준다. 거기에는 몇가지 주의할것들이 있다. find연산은 2진탐색나무에서 탐색값 x에 일치하는 항목에 대한 (상수적인)참조를 되돌린다. 일치검색은 특별한 Comparable형에 대하여 정의되여야 하는 <연산자에 기초한다. 특히 x<y와 y<x가 둘다 거짓이면 항목 x는 y와 일치한다. 이것은 형의 일부분(사회안전번호자료성원이나 생활비와 같은)에 대해서만 정의된 비교함수를 가지는 복잡한 형(종업원기록과 같은)도 Comparable로 되게 한다. 제1장 제6절 3에서 Comparable로 리용될수 있는 클라스를 설계하는 일반적인 수법을 설명하였다.

```
template <c1ass Comparab1e>
class BinarySearchTee
{
 public:
explicit Bi'narySearchTreeC const Comparable & notFound );
B-inarySearchTree( const Bi'narySearchTree & rhs );
-BinarySearchTreeC );

const Comparable & findMin( ),const;
const Comparable & f-indMax( ) const;
const Comparable & f"ind( const Comparable & x ) const;
bool isEmpty( ) const;
void printTree( ) const;

void makeEmptyC );
void insert( const Comparable & x );
void remove( const Comparable & x );

const BinarySearchTree & operator=( const BinarySearchTree & rhs );

private:
B-inaryNode<Comparab1e> "root;
const Comparable ITEM_NOT_FOUND;

const Comparable & e1ementAt( BinaryNode<Comparab1e> *t ) const;

void insertC const Comparable & x, B"inaryNode<Comparab1e> * & t ) const;
  void remove( const Comparable & x, BinaryNode<Comparab1e> * & t ) const;
  B1naryNode<Comparab1e> * findM-inC BinaryNode<Comparab1e> '-"t )
```

```
        BinaryNode<Comparable> * find( const Comparable & x,
                                       BinaryNode<Comparable> *t ) const;
    void makeEmpty( BinaryNode<Comparable> * & t ) const;
    void printTree( BinaryNode<Comparable> *t ) const;
    BinaryNode<Comparable> * clone( BinaryNode<Comparable> *t ) const;
};
```

프로그램 4-6. 2진탐색나무에 대한 클라스골격

한가지 중요한 문제는 find연산이 실패하면 어떻게 처리하겠는가를 결정하는것인데 여기에는 다음과 같이 여러가지 방안이 있다.

- find는 례외를 발생한다.
- find는 참조변수로써 넘겨 지는 bool형파라메터를 추가로 가진다. 이 파라메터는 find가 성공하였는가를 지적한다.
- find는 특수한 ITEM_NOT_FOUND값을 되돌린다.

여기서는 세번째 방안을 선택하자. ITEM_NOT_FOUND는 BinarySearchTree클라스의 추가적인 자료성원으로서 그 값은 구축자에서 초기화된다. 그것은 한번 초기화되면 더는 수정할수 없는 const자료성원이다.

다른 자료성원은 뿌리매듭에 대한 지적자인데 이 지적자는 빈나무들에 대하여서는 NULL이다. 공개성원함수들은 비공개재귀함수들을 호출하기 위한 일반적인 수법을 리용한다. 이것이 find를 어떻게 진행하는가 하는 실례는 프로그램 4-7에서 보여 준다. 한행의 코드를 차지하는 시끄러운 형판문법으로 하여 클라스대면부에 이러한 성원함수들이 많게 되는것은 이상하지 않다. 비공개함수 elementAt는 t로 지적된 매듭에 보관된 항목에 대한 상수적인 참조 또는 t가 NULL일 때 ITEM_NOT_FOUND를 되돌린다. 대부분의 공개성원함수들에서 이와 류사한 수법들이 리용되므로 그 코드는 반복하지 않는다.

```
/**
 * Find item x in the tree.
 * Return the matching item or ITEM_NOT_FOUND if not found.
 */
template <class Comparable>
const Comparable & BinarySearchTree<Comparable>::
find( const Comparable & x ) const
{
    return elementAt( find( x, root ) );
}
/**
 * Internal method to get element data member in node t.
 * Return the element data member or ITEM_NOT_FOUND if t is NULL.
 */
```

```
template <c1ass Comparab1e>
const Comparable & BinarySearchTree<Comparab1e>::
e1ementAt( BinaryNode<Comparab1e> *t ) const
{
      return t == NULL ? ITEM_NOT_FOUND : t->e1ement;
}
```

프로그람 4-7. 재귀적인 비공개성원함수를 호출하는
공개부성원함수에 대한 설명

몇개의 비공개성원함수들은 참조에 의한 호출을 리용하여 지적자값을 넘기는 수법을 리용한다. 이것은 공개성원함수들이 뿌리에 대한 지적자를 재귀적인 비공개성원함수들에 넘길수 있게 한다. 이때 재귀함수들은 뿌리가 또 다른 매듭을 가리키도록 뿌리의 값을 수정할수 있다. insert에 대한 코드를 검사할 때 더 자세한 설명을 하기로 한다. 여기서는 몇가지 비공개산법들을 서술한다.

1. find

이 연산은 나무 *T*에서 항목 *X*를 가지는 매듭에 대한 지적자를 되돌리거나 그런 매듭이 없으면 NULL을 되돌릴것을 요구한다. 나무구조는 이것을 간단히 처리한다. 만일 *T*가 비였다면 NULL을 되돌릴수 있다. 한편 *T*에 보관된 항목이 *X*라면 *T*를 되돌릴수 있다. 만일 그렇지 않으면 *X*와 *T*에 보관된 항목의 관계에 따라 *T*의 왼쪽이나 오른쪽 부분나무에 대하여 재귀적인 호출을 진행한다. 프로그람 4-8은 이 방법을 실현한 코드를 보여 준다.

```
/**
 * Internal method to find an item in a subtree.
 * x is item to search for.
 * t is the node that roots the tree.
 * Return node containing the matched item.
 */
template <c1ass Comparab1e>
BinaryNode<Comparab1e> *
BinarySearchTree<Comparab1e>::
find( const Comparable & x, BinaryNode<Comparab1e> *t ) const
{
      if( t == NULL )
          return NULL;
      else if( x < t->e1ement )
          return find( x, t->1eft );
      else if( t->e1ement < x )
```

```
        return find( x, t->right );
    else
        return t;       // Match
}
```

프로그람 4-8. 2진탐색나무에 대한 find연산

검사순서에 주의하시오. 먼저 주어 진 나무가 빈나무인가를 검사하는것이 아주 중요하다. 만일 그렇지 않으면 NULL지적자를 가지고 자료성원에 접근하려고 시도하여 실행시 오유가 발생하기때문이다. 나머지 검사는 최소한도의 있을수 있는 경우에 대하여 진행된다. 또한 두개의 재귀적호출들은 실제로 하한재귀들이며 while순환고리로 쉽게 이전할수 있다는데 주의해야 한다. 하한재귀를 리용하면 간단한 알고리듬으로 속도의 감소를 보상할수 있으므로 매우 적합한것이며 리용되는 탄창공간은 단지 $O(\log N)$으로 될것이다.

2. findMin과 findMax

이 비공개루틴들은 각각 나무에서 가장 작은 요소와 가장 큰 요소를 포함하는 매듭에 대한 지적자를 되돌린다. findMin을 실행하기 위해서는 뿌리에서 시작하여 왼쪽 자식을 따라서 왼쪽으로 계속 내려 간다. 정지하는 위치는 가장 작은 요소이다. findMax루틴은 분기가 오른쪽 자식으로 된다는것을 제외하면 findMin과 같다.

이것은 매우 쉬우므로 많은 프로그람작성자들이 재귀의 리용을 즐긴다. 이제 findMin은 재귀적으로, findMax는 비재귀적으로 처리하는 두가지 방법의 루틴들을 코드로 준다(프로그람 4-9, 4-10를 보시오.).

```
/**
 * Internal method to find the smallest item in a subtree t.
 * Return node containing the smallest item.
 */
template <c1ass Comparab1e>
BinaryNode<Comparab1e> *
BinarySearchTree<Comparab1e>::findMin(  BinaryNode<Comparab1e>  *t  )
const
{
    if( t == NULL )
        return NULL;
    if( t->left == NULL )
        return t;
    return findMin( t->left );
}
```

프로그람 4-9. 2진탐색나무에 대한 findMin의 재귀적인 실현

```
/**
  * Internal method to find the largest item in a subtree t.
  * Return node containing the largest item.
   */
template <c1ass Comparab1e>
BinaryNode<Comparab1e> *
  BinarySearchTree<Comparab1e>::findMax( BinaryNode<Comparab1e> *t ) const
{
    if( t != NULL )
        whi1e( t->right != NULL )
            t = t->right;
    return t;
}
```

프로그람 4-10. 2진탐색나무에 대한 findMax의 비재귀적인 실현

find연산에서는 또한 빈나무의 퇴화경우를 어떻게 처리하겠는가에 주의를 돌려야 한다. 이것은 언제나 중요하지만 재귀프로그람들에서는 특별히 더 중요하다. 또한 지적자의 복사를 가지고 작업할뿐이므로 findMax에서 t를 변경하는것이 안전하다는것에 주목해야 한다. 그러나 t->right = t->right->right와 같은 명령들은 변경을 가져 올수 있기때문에 특별히 주의하여야 한다.

3. insert

삽입루틴은 개념적으로 간단하다. 나무 *T*에 *X*를 삽입하기 위하여 find로써 나무를 훑는다. 만일 *X*를 찾으면 아무러한 처리도 하지 않는다(또는 일부 《갱신》한다.). 만일 찾지 못하면 순회경로상의 마지막위치에 *X*를 삽입한다. 그림 4-12는 이 과정을 보여 준다. 5를 삽입하기 위하여 find를 수행하면서 나무를 순회한다. 항목 4를 가진 매듭에서 오른쪽으로 가야 하지만 거기에는 부분나무가 없다. 즉 5는 그 나무에 없는것으로 되며 이것은 5를 삽입하기 위한 정확한 위치로 된다.

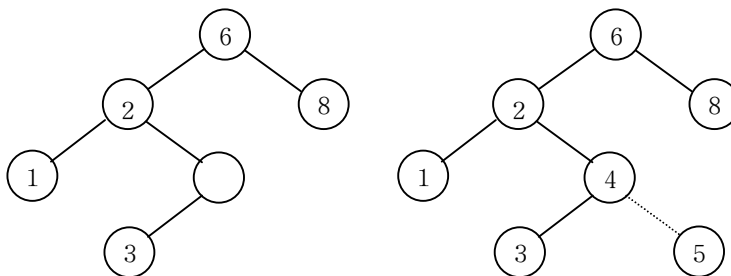



그림 4-12. 5를 삽입하기전과 삽입한후의 2진탐색나무

중복현상은 매듭기록에 발생빈도를 지적하는 여분마당을 유지하는것으로서 조종될수 있다. 이것은 전체 나무에 약간의 여분공간을 추가하지만 나무에 대하여 반복적인 처리를 하는것보다는 더 좋다(이것은 나무를 아주 깊게 만드는 경향이 있다.). 물론 이 방법은 <연산자를 위한 열쇠가 큰 자료구조에서 국부적이라면 효과가 적다. 이런 경우에는 목록이나 또 다른 탐색나무와 같은 보조자료구조에서 같은 열쇠를 가지는 모든 구조체들을 유지할수 있다.

프로그람 4-11은 삽입루틴에 대한 코드를 보여 준다. 4행과 6행은 적당한 부분나무에 x를 재귀적으로 삽입하여 첨부한다. 이 재귀루틴에서 t는 새로운 잎매듭이 만들어 질 때에만 변경된다. 이때 재귀루틴은 그 잎매듭의 부모인 다른 매듭 p로부터 재귀적으로 호출되였다는것을 의미한다. 그 호출은 insert(x, p->left) 또는 insert(x, p->right)일것이다. 두 호출에서 t는 현재 p->left 또는 p->right에 대한 참조인데 이것은 p->left나 p->right가 새로운 매듭을 가리키도록 변경된다는것을 의미한다. 전체적으로 이것은 재치 있는 수법이다.

```
/**
 * Internal method to insert into a subtree.
 * x is the item to insert.
 * t is the node that roots the tree.
 * Set the new root.
 */
template <c1ass Comparable>
void BinarySearchTree<Comparab1e>::
insert( const Comparable & x, BinaryNode<Comparable> * & t ) const
{
      if( t == NULL )
          t = new BinaryNode<Comparab1e>( x, NULL, NULL );
      else if( x < t->e1ement )
          insert( x, t->1eft );
      else if( t->e1ement < x )
          insert( x, t->right );
      else
          ;    // Duplicate; do nothing
}
```

프로그람 4-11. 2진탐색나무에로의 삽입

4. remove

많은 자료구조들에서 공통적으로 가장 어려운 연산은 삭제이다. 일단 삭제될 매듭을 찾으면 여러가지 가능성을 고려하여야 한다.

만일 그 매듭이 잎매듭이면 그것은 즉시 삭제될수 있다. 그리고 삭제될 매듭이 하나의 자식을 가지면 그 매듭의 부모가 삭제될 매듭을 우회하여 련결을 조정한 다음에 그 매듭을 삭제한다(정확성을 위하여 련결방향을 명백히 표시한다.). 그림 4-13을 보시오.

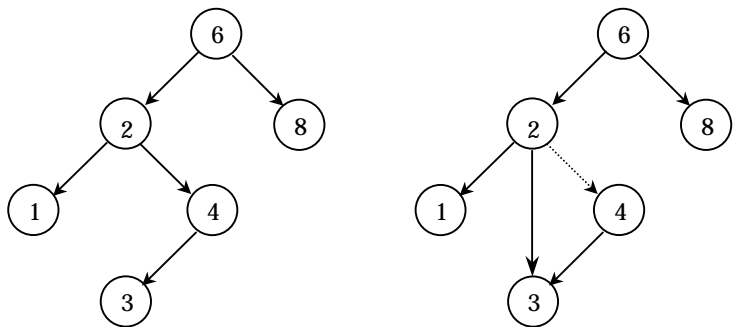



그림 4-13. 한개의 자식을 가지는 매듭(4)의 삭제전과 삭제후

복잡한 경우는 2개의 자식을 가지는 매듭에 대해서이다. 일반적인 방법은 이 매듭의 자료를 오른쪽 부분나무의 가장 작은 자료로 치환하고(이것은 쉽게 찾을수 있다.) 그 매듭을 재귀적으로 삭제하는것이다(이것은 현재 빈것이다.). 오른쪽 부분나무에 있는 가장 작은 매듭이 왼쪽 자식을 가질수 없기때문에 두번째 remove는 쉽다. 그림 4-14는 초기나무와 삭제결과를 보여 준다. 삭제되여야 할 매듭은 뿌리의 왼쪽 자식인데 열쇠값은 2이다. 그것은 그의 오른쪽 부분나무의 가장 작은 자료 3으로 치환되고 그다음에 종전과 마찬가지로 그 매듭을 삭제한다.

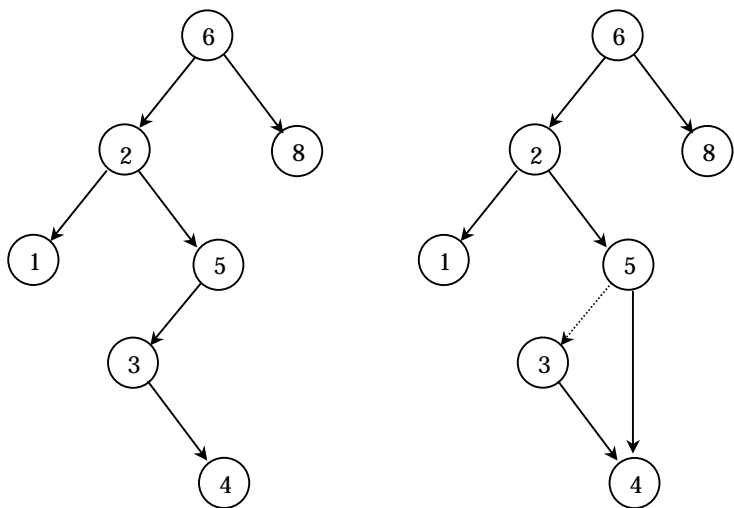



그림 4-14. 2개의 자식들을 가지는 매듭(2)의 삭제전과 삭제후

프로그람 4-12의 코드는 삭제산법을 수행한다. 그것은 이것이 적당할 때 오른쪽 부분나무에 있는 가장 작은 매듭을 찾아서 삭제하기 위하여 나무를 두번 훑기때문에 불합리하다. 특별한 removeMin산법을 서술하여 이 불합리성을 제거할수 있으나 여기서는 간단하게 취급하기 위하여 그것을 무시한다.

만일 삭제회수가 적을것이 기대되면 그때 리용하는 일반적인 방법은 **지연삭제**(lazy deletion)이다. 즉 어떤 요소가 삭제될 때 그것은 다만 삭제표식을 붙여 나무에 남겨 놓는다. 이 방법은 그때 출현빈도수를 보관하는 자료성원이 감소될수 있으므로 중복항목이 존재하는 경우에 매우 유리하다. 만일 나무에서 실재하는 매듭들의 수가《삭제된》매듭수와 같으면 나무의 깊이는 매우 작은 상수량만큼(왜?) 증가만 할것이며 따라서 지연삭제와 관련한 시간초과는 매우 작다. 또한 삭제된 항목이 다시 삽입되면 새로운 세포를 할당하는 공정은 필요 없게 된다.

```
/**
 * Internal method to remove from a subtree.
 * x is the item to remove.
 * t is the node that roots the tree.
 * Set the new root.
 */
template <class Comparable>
void BinarySearchTree<Comparable>::
remove( const Comparable & x, BinaryNode<Comparable> * & t ) const
{
    if( t == NULL )
        return;     // Item not found; do nothing
    if( x < t->element )
        remove( x, t->left );
    else if( t->element < x )
        remove( x, t->right );
    else if( t->left != NULL && t->right != NULL ) // Two children
    {
        t->element = findMin( t->right )->element;
        remove( x, t->right );
    }
    else
    {
        BinaryNode<Comparable> *oldNode = t;
        t = ( t->left != NULL ) ? t->left : t->right;
        delete oldNode;
    }
}
```

프로그람 4-12. 2진탐색나무에서 삭제루틴

5. 해체자와 복사대입연산자

일반적으로 해체자는 makeEmpty를 호출한다. 아직 고찰되지 않은 공개부의 makeEmpty루틴은 간단히 비공개부의 재귀적인 형식을 호출한다. 프로그람 4-13에서 보여 주는것처럼 t의 자식들을 재귀적으로 처리한 다음에 delete에 의해서 t를 해방한다. 따라서 모든 매듭들이 재귀적으로 갱신된다. 마지막에 t 즉 root는 NULL지적자로 수정된다. 프로그람 4-14에서 보여 주는 복사대입연산자는 일반적인 절차에 따르는데 먼저 makeEmpty를 호출하여 어떤 기억공간을 갱신하도록 하고 그다음 rhs의 복사를 처리한다. clone이라고 하는 재귀함수를 리용하면 지저분한 모든 처리를 재치 있게 수행한다.

```
/**
  * Destructor for the tree.
 */
template <c1ass Comparab1e>
BinarySearchTree<Comparab1e>::~BinarySearchTree( )
{
      makeEmpty( );
}

/**
 * Internal method to make subtree empty.
 */
template <class Comparab1e>
void BinarySearchTree<Comparab1e>::
makeEmpty( BinaryNode<Comparab1e> * & t ) const
{
      if( t != MULL )
      {
          makeEmpty( t->left );
          makeEmpty( t->right);
          delete t;
      }
    t = NULL;
}
```

프로그람 4-13. 해제자와 재귀적인 makeEmpty성원함수

```
/**
 * Deep copy.
 */
template <c1ass Comparab1e>
const BinarySearchTree<Comparab1e> &
BinarySearchTree<Comparab1e>::
```

```
 operator=( const BinarySearchTree<Comparable> & rhs )
 {
     if( this != &rhs )
     {
         makeEmpty( );
         root = clone( rhs.root );
     }
     return *this;
}

/**
  * Internal method to clone subtree.
  */
template <class Comparable>
BinaryNode<Comparable> *
BinarySearchTree<Comparable>::clone( BinaryNode<Comparable> * t ) const
{
     if( t == NULL )
         return NULL;
     else
          return new BinaryNode<Comparable>( t->element, clone( t->left ),
                                                        clone( t->right ) );
}
```

프로그람 4-14. operator=연산자와 재귀적인 clone성원함수

6. 평균경우분석

직관적으로 makeEmpty와 operator=를 제외하고 앞절에 있는 모든 연산들은 $O(\log N)$시간을 가지리라고 생각되는데 상수시간안에 나무에서 한준위만큼 내려 오기때문에 그때 나무에 대한 연산은 크기가 대략 절반까지는 줄어 들게 된다. 실제로 모든 연산들(makeEmpty와 operator=를 제외하고)의 실행시간은 $O(d)$(여기서 d는 접근된 항목을 포함하는 매듭의 깊이)이다.

여기서는 어떤 나무에 대하여 모든 삽입서렬들이 거의 동등하다고 가정하고 모든 매듭들의 평균깊이가 $O(\log N)$라는것을 증명한다.

나무에서 모든 매듭들의 깊이의 합을 **내부경로길이**(internal path length)라고 한다. 이제 2진탐색나무의 평균적인 내부경로길이를 계산해 보자. 여기서 평균은 2진탐색나무에 대한 모든 가능한 삽입서렬들에 대하여 취해 진다.

$D(N)$이 N개의 매듭을 가지는 어떤 나무 T에 대한 내부경로길이라고 하자. $D(1)=0$이다. N개의 매듭을 가진 나무는 $0 \le i \le N$에 대하여 i개의 매듭을 가진 왼쪽 부분나무와 $(N - i - 1)$개의 매듭을 가진 오른쪽 부분나무에 깊이가 령인 뿌리를 더한것으로 구성된다.

$D(i)$는 뿌리를 고려한 왼쪽 부분나무의 내부경로길이이다. 기본나무에서 모든 매듭들은 한 준위 더 깊다. 오른쪽 부분나무에 대해서도 역시 같다. 따라서 그것을 재현할수 있다.

$$D(N) = D(i) + D(N - i - 1) + N - 1$$

만일 모든 부분나무들의 크기가 거의 같다면(그 부분나무들은 크기가 나무에 삽입된 첫 요소의 상대적인 위수에만 관계되기때문에 2진탐색나무로는 되지만 2진나무는 아니다.) 그때 $D(i)$와 $D(N - i - 1)$의 평균값은 $(1/N)\sum_{j=0}^{N-1} D(j)$이다. 이것은

$$D(N) = \frac{2}{N}\left[\sum_{j=0}^{N-1} D(j)\right] + N - 1$$

로 계산된다. 이 식은 $D(N)=O(N\log N)$의 평균값을 가지는데 제7장에서 다시 설명되여 해결될것이다. 따라서 어떤 매듭의 예상되는 깊이는 $O(\log N)$이다. 실레로 그림 4-15에서 보여 준 우연적으로 발생된 500개의 매듭을 가지는 나무는 그 깊이가 9.98로 예상된다.

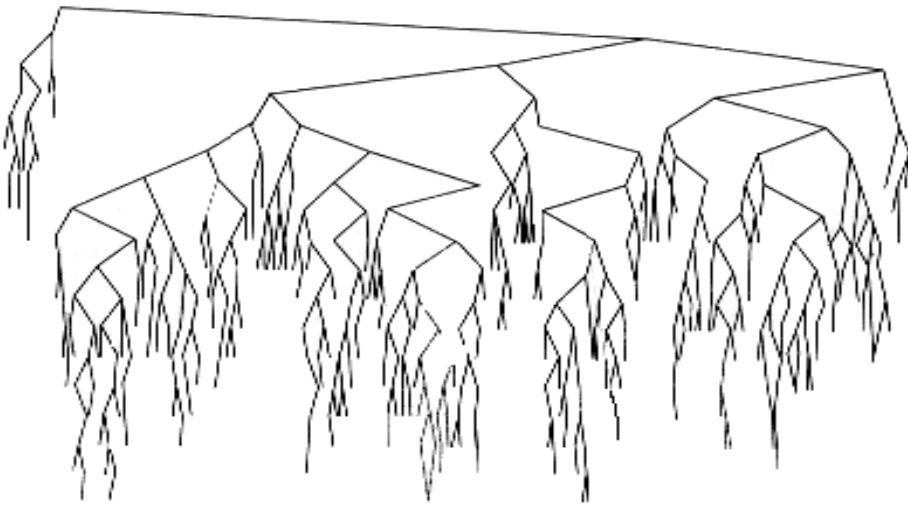



그림 4-15. 우연적으로 발생된 2진탐색나무

이 결과는 앞절에서 설명된 모든 연산들의 평균실행시간이 $O(\log N)$이지만 언제나 그런것은 아니라는것을 의미한다. 그 리유는 삭제연산들때문에 모든 2진탐색나무들이 언제나 같다고 명백히 말할수 없기때문이다. 실제로 우에서 서술된 삭제알고리듬은 항상 삭제되는 매듭을 오른쪽 부분나무에 있는 매듭으로 치환하기때문에 왼쪽 부분나무가 오른쪽 부분나무보다 더 깊어 지게 된다. 이 방법의 정확한 효과는 아직 알려 지지 않았으며 이것은 다만 리론상의 착상으로 될뿐이다. 만일 삽입들과 삭제들이 $\Theta(N^2)$번 반복된다면 나무의 깊이는 $\Theta(\sqrt{N})$으로 예상된다. 25만번의 우연적인 insert/remove쌍이 실행되면 나

무는 그림 4-15에서와 같이 어느 정도 오른쪽이 무거워 져 틀림없이 균형이 이루어 지지 않는다(평균깊이는 12.51). 그림 4-16을 보시오.

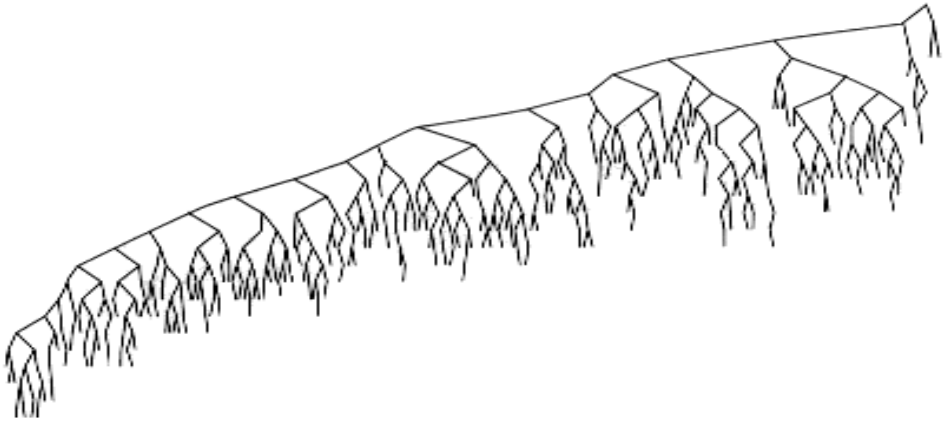



그림 4-16. $\Theta(N^2)$개의 insert/remove쌍들이 처리된 다음의 2진탐색나무

여기서는 삭제될 요소를 치환할 때 오른쪽 부분나무에 있는 가장 작은 요소와 왼쪽 부분나무에 있는 가장 큰 요소사이에서 우연적으로 선택하여 그 문제를 해결하여야 한다. 이것은 얼핏 보기에는 편중을 없애고 균형을 유지하면 될것 같지만 아직까지 누구도 이것을 해결하지 못하였다. 어쨌든 이 현상은 작은 나무들에서는 효과가 없고 여전히 습관되지 않았으며 $O(N^2)$의 insert/remove쌍들이 리용되면 나무는 균형을 얻은것처럼 보이기때문에 리론상의 착상으로 된다.

론의의 초점은 《평균》의 의미를 결정하는것이 일반적으로 대단히 어려우며 유효할수도 있고 유효하지 않을수도 있는 전제를 요구할수 있다는것이다. 삭제연산이 없거나 지연삭제가 리용될 때는 우의 연산들의 평균실행시간이 $O(\log N)$이라는것을 판단할수 있다. 우에서 설명된것과 같이 특이한 경우를 제외하고 이 결과는 고찰된 성질들과 대단히 잘 일치한다.

만일 미리 정렬된 나무에 입력이 주어 지면 그 나무는 왼쪽 자식이 없는 매듭들만으로 이루어 지므로 련속적인 insert연산들은 2차원적인 시간을 가지며 련결목록의 실현은 매우 비경제적일것이다. 이 문제에 대한 한가지 해결방안은 균형(balance)이라고 하는 여분의 구조적조건을 주는것이다. 즉 매듭이 너무 깊어 지는것은 허용하지 않도록 한다.

균형나무를 실현하기 위한 일반적인 알고리듬들이 몇가지 있다. 그 대부분은 표준적인 2진탐색나무보다 아주 복잡하며 갱신시간은 평균적으로 더 길다. 그러나 그것들은 극단적으로 간단한 경우들로부터 보호한다. 뒤에서는 균형탐색나무의 가장 오래된 형태의 하나인 AVL나무에 대하여 간단히 고찰한다.

다음으로 보다 새로운 산법은 균형조건을 미리 알고 임의의 깊이를 가지는 나무에 대하여 앞으로의 연산들을 효과적으로 처리하도록 매개 연산후에 나무를 재구축하는 규칙을 적용하는것이다. 이러한 형식의 자료구조들은 일반적으로 자동조정식(self-adjusting)으로 분류된다. 2진탐색나무의 경우에 임의의 한개 연산에 대하여 $O(\log N)$한계를 담보할수 없지만 M개의 연산들의 서렬은 최악의 경우에 총 $O(M\log N)$의 실행시간을 가진다. 이것이면 일반적으로 최악의 경우에 대한 보호에는 충분하다. 펼친나무(splay tree)라는 자료구조가 있는데 그 분석은 어지간히 복잡하며 앞으로 제11장에서 설명하게 된다.

제4절. AVL나무

AVL나무(Adelson-Velskii와 Landis에 의해서 제안된 나무형태의 자료구조)는 균형조건을 가지는 2진탐색나무이다. 균형조건은 유지하기가 쉬워야 하며 나무의 깊이가 $O(\log N)$이라는것을 담보해야 한다. 그의 가장 간단한 사상은 왼쪽과 오른쪽 부분나무들이 같은 높이를 가지도록 하는것이다. 그림 4-17에서 보여 주는것처럼 이 사상은 나무의 깊이가 억지로 얕아 지도록 하지는 않는다.

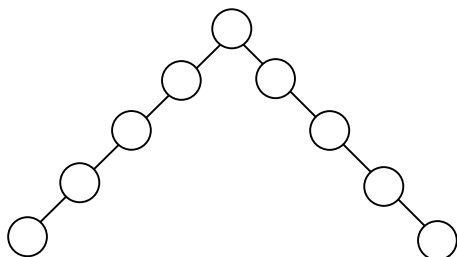



그림 4-17. 좋지 못한 2진나무. 뿌리에서 평형을 요구하는것은 충분하지 못하다.

또 다른 균형조건은 매 매듭의 왼쪽과 오른쪽 부분나무들이 같은 높이를 가지도록 하는것이다. 만일 빈 부분나무의 깊이를 -1로 정의하면 2^k-1개의 매듭들을 가지는 완전균형2진나무만이 이 조건을 만족시킨다. 따라서 이것이 깊이가 작은 나무들을 보증한다고 하더라도 균형조건이 너무 엄격하여 리용하기 어려우며 완화시켜야 할 필요가 있다.

AVL나무는 나무의 매개 매듭에 대하여 왼쪽과 오른쪽 부분나무의 높이가 많아서 1만큼 차이날수 있다는것을 제외하고는 2진탐색나무와 같다(빈나무의 깊이는 -1로 정의된다.). 그림 4-18에서 왼쪽에 있는 나무는 AVL나무이지만 오른쪽에 있는 나무는 AVL나무가 아니다. 높이정보는 매 매듭에 대하여 유지된다(매듭구조체에서). AVL나무의 높이는 최대로 $1.44\log(N+2)-0.328$정도이지만 실제로는 $\log N$보다 약간 더 크다. 실례로 그림 4-19에 가장 적은 수의 매듭(143개)들을 가진 높이가 9인 AVL나무를 보여 준다. 이 나무는 왼쪽 부분나무로서 최소높이 7인 AVL나무를 가진다. 오른쪽 부분나무는 최소높이

8인 AVL나무이다. 이것은 높이가 h인 AVL나무에서 최소매듭수 $S(h)= S(h\text{-}1)+S(h\text{-}2)+1$이라는것을 의미한다. h=0이면 $S(h)$=1이고 h=2이면 $S(h)$=2이다. 함수 $S(h)$는 피보나치수들과 상당한 관련이 있으며 이로부터 AVL나무의 높이에 대하여 우에서 설명한 한계가 얻어진다.

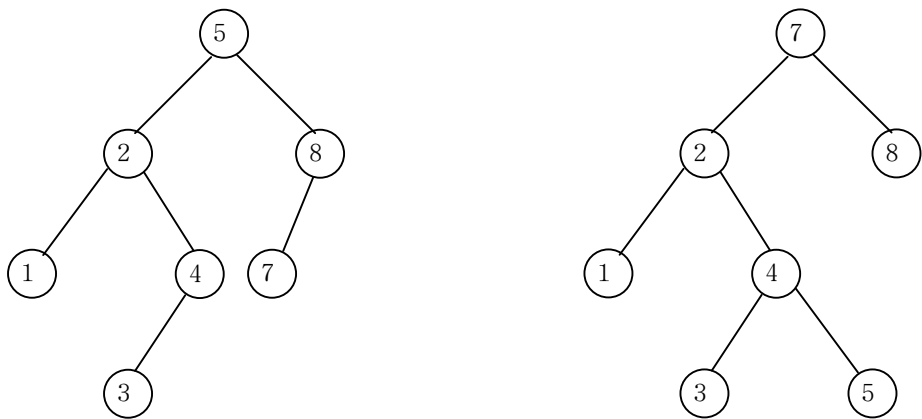



그림 4-18. 두개의 2진탐색나무들. 왼쪽 나무만이 AVL나무이다.

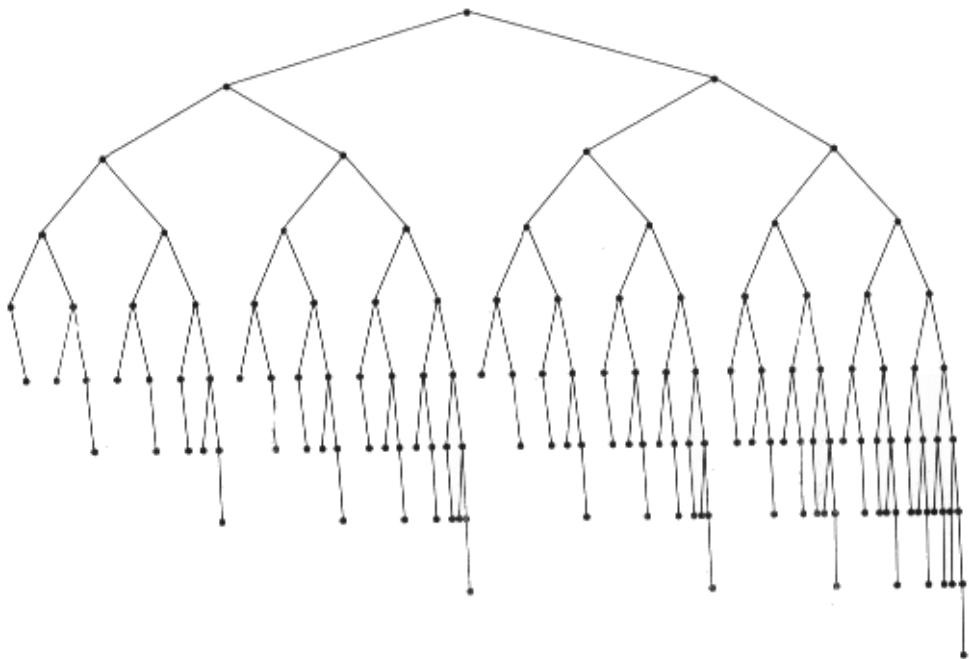



그림 4-19. 높이가 9인 가장 작은 AVL나무

따라서 나무연산들은 삽입을 제외하고는 모두 $O(\log N)$시간에 실행될수 있다(여기에

서는 지연삭제를 리용하게 된다.). 삽입할 때 뿌리매듭까지의 경로상에 있는 매듭들의 균형정보를 전부 수정하는것이 필요하며 삽입이 잠재적으로 어렵게 되는 리유는 매듭을 삽입할 때 AVL나무의 성질이 파괴될수 있기때문이다(실례로 그림 4-18에 있는 AVL나무에 6을 삽입할 때 열쇠 8을 가진 매듭에서 균형조건이 파괴된다.). 이런 경우에 그 성질은 삽입단계가 계속되기전에 수복되여야 한다. 그것은 회전(rotation)이라고 하는 간단한 수법을 나무에 적용하여 실현할수 있다.

삽입후에 삽입위치로부터 뿌리까지의 경로상에 있는 매듭들만이 변경된 부분나무를 가지기때문에 그 매듭들에서만 균형이 변하게 된다. 뿌리로 향한 경로를 따라 가면서 균형정보를 갱신하므로 AVL조건을 위반하는 매듭을 찾을수 있다. 맨 처음의 그러한 매듭(즉 가장 깊은)에서 나무를 어떻게 재균형맞추겠는가를 고찰하고 이러한 재균형이 전체나무에 대하여 AVL성질을 만족시킨다는것을 증명한다.

재균형되여야 할 매듭 α를 호출하자. 임의의 매듭이 많아서 2개의 자식을 가지며 또한 α의 두개 부분나무들의 높이가 2만큼 차이나면 높이불균형이므로 균형위반이 4가지 경우에 발생된다는것을 알수 있다.

① α의 왼쪽 자식의 왼쪽 부분나무에 삽입
② α의 왼쪽 자식의 오른쪽 부분나무에 삽입
③ α의 오른쪽 자식의 왼쪽 부분나무에 삽입
④ α의 오른쪽 자식의 오른쪽 부분나무에 삽입

①과 ④의 경우는 α를 고려한 **경상대칭**(평면거울의 반사에 의하여 만들어 진 어떤 물체의 영상이라는 뜻으로서 사물 또는 현상, 조건이 거울에 반사된것처럼 대칭된다는 의미)이고 ②와 ③의 경우도 이와 마찬가지이다. 그 결과 리론적인 문제로서 두가지 기본경우가 있다. 물론 프로그람작성의 호상관계로부터 여기에는 여전히 4가지 경우가 존재한다.

첫째로, 삽입이 나무의 《바깥쪽》(즉 왼쪽-왼쪽 또는 오른쪽-오른쪽)에서 발생하는 경우에는 나무의 단일회전(single rotation)으로 처리할수 있다. 둘째로, 삽입이 《안쪽》(즉 왼쪽-오른쪽 또는 오른쪽-왼쪽)에서 발생하는 경우에는 좀 더 복잡한 2중회전(double rotation)으로 처리할수 있다. 이것들은 나무에 대한 기본적인 연산들로서 균형나무에 대한 알고리듬들에서 자주 리용된다. 이 절의 뒤에서는 이 루틴들을 설명하고 그것들이 균형을 유지하는데 충분하다는것을 증명하며 AVL나무의 일반적인 실현을 준다. 제12장에서 다른 균형나무산법들에 대하여 더 구체적으로 고찰한다.

1. 단일회전

그림 4-20은 첫번째 경우를 설명하는 단일회전을 보여 준다. 왼쪽의 그림은 회전하

기전이고 오른쪽의 그림은 회전한 다음이다. 주의 깊게 분석하자. 매듭 k_2은 그의 왼쪽 부분나무가 오른쪽 부분나무보다 준위가 2만큼 더 깊기때문에 AVL균형성질을 위반한다 (그림에서 중간에 있는 점선들은 준위를 표시한다.). 표현된 상태는 삽입전에 k_2가 AVL 성질을 만족하도록 조종하는 첫번째 경우에 대한 유일한 방안이지만 그것은 새로운 매듭을 삽입한후에는 균형조건을 또 위반하게 된다. 부분나무 X는 Z보다 정확히 2준위 더 깊기때문에 특별한 준위로 된다. Y는 k_2가 삽입후에 균형성을 잃었기때문에 새로운 X와 같은 준위에 있을수 없다. 그리고 Y는 또한 k_1이 AVL평행조건을 위반한 뿌리매듭으로 향하는 경로상의 첫번째 매듭이기때문에 Z와 같은 준위일수 없다.

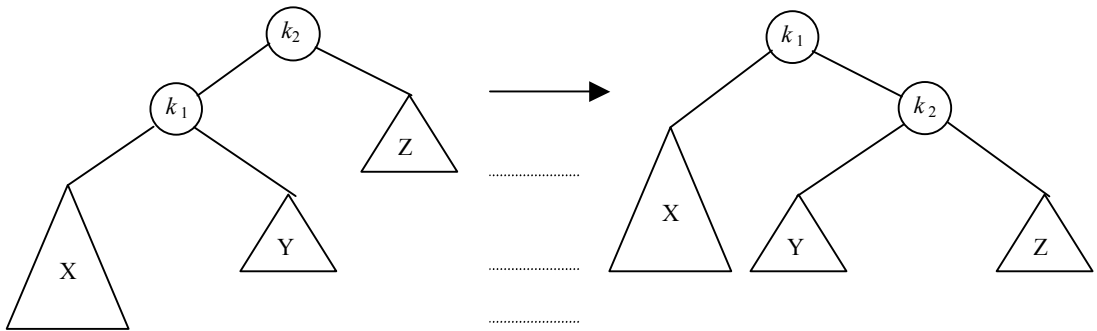



그림 4-20. 첫번째 경우를 조정하는 단일회전

나무를 리론적으로 다시 균형으로 만들기 위하여서는 X를 우로 한준위, Z를 아래로 한준위 이동하여야 한다. 이것은 실제로 AVL성질이 요구하는것보다 더 좋은것이다. 이것을 수행하기 위하여 그림 4-20의 두번째 부분에서 보여 준것과 같이 매듭들을 다시 정리한다. 여기에는 추상적인 방안이 있다. 즉 다루기 쉽게 나무를 시각화하고 자식매듭 k_1을 끌어 올려서 그것을 분기시키고 Y를 k_2에 련결한다. 그 결과 k_1이 새로운 뿌리로 된다. 2진탐색나무의 성질은 처음의 나무에서 $k_2 > k_1$이므로 k_2는 새로운 나무에서 k_1의 오른쪽 자식으로 된다. X와 Z는 각각 k_1의 왼쪽 자식과 k_2의 오른쪽 자식으로 남아 있다. 부분나무 Y는 처음 나무에서 k_1과 k_2사이에 있는 항목들을 가지고 있는데 새로운 나무에서는 k_2의 왼쪽 자식으로 배치될수 있다. 이것들은 모두 순서적인 요구를 만족시킨다.

적은 수의 지적자들만을 변경할것을 요구하는 이러한 처리결과로써 AVL나무인 또 하나의 2진탐색나무를 만든다. 이것은 X가 한준위 우로 이동하고 Y는 같은 준위에 남아 있으며 Z가 한준위 아래로 이동하는것으로 된다. k_2와 k_1은 AVL요구를 만족시킬뿐아니라 정확히 같은 깊이를 가지는 부분나무도 가진다. 더우기 전체 부분나무의 새로운 높이는 정확히 X를 증가시킨 삽입하기전 단계에서의 초기부분나무와 같다. 따라서 뿌리에로의 경로상에서 높이들은 더 수정할 필요가 없으며 론리적으로 그이상의 회전은 필요 없다.

그림 4-21의 왼쪽 그림에서는 초기AVL나무에 6을 삽입한 다음 매듭 8의 균형조건이 파괴되는것을 보여 준다. 따라서 7과 8사이에서 단일회전을 실시하여 오른쪽 그림과 같은 나무를 얻는다.

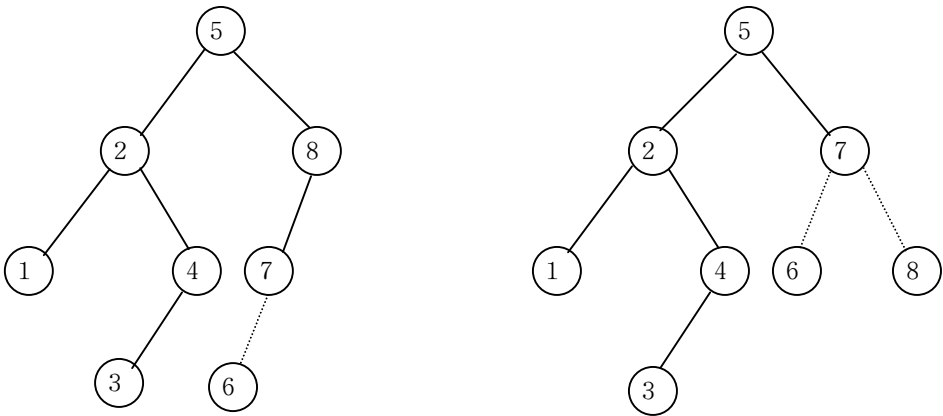



그림 4-21. 6의 삽입으로 AVL성질이 파괴. 그때 단일회전으로 조정

이미 언급된것처럼 4번째 경우는 대칭적인 경우를 표현한다. 그림 4-22는 단일회전이 어떻게 적용되는가를 보여 준다. 좀 더 구체적인 실례를 가지고 설명하자. 초기에 빈 AVL나무를 가지고 시작하여 3, 2, 1항목들을 삽입하고 그다음 4부터 7까지 순차적으로 삽입한다고 하자. 첫 문제는 항목 1을 삽입할 때 발생하는데 그것은 뿌리에서 AVL성질이

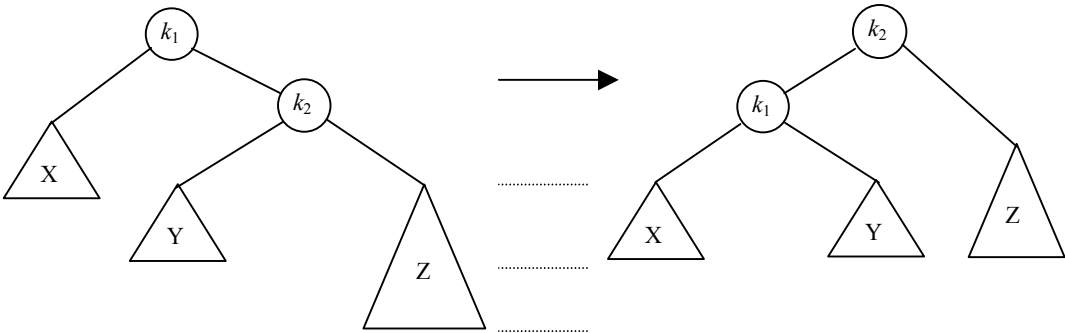
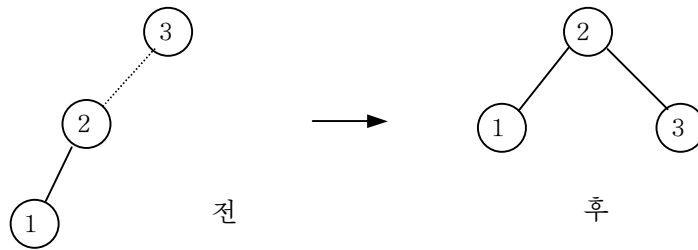
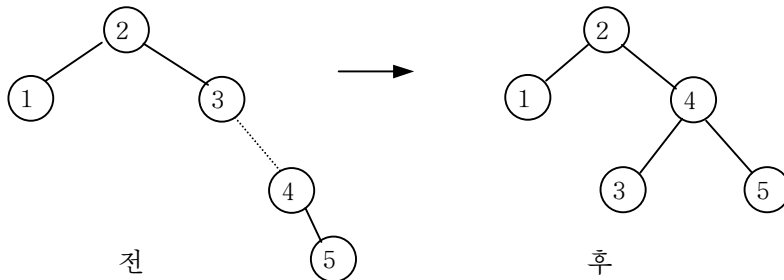
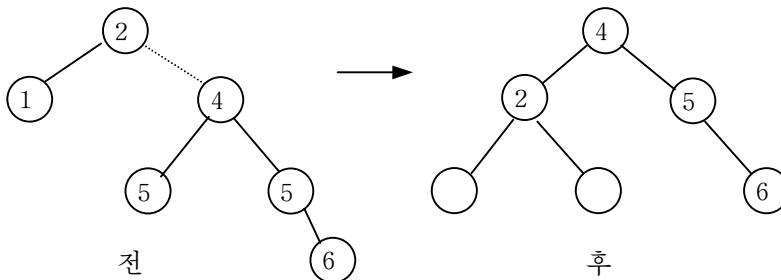
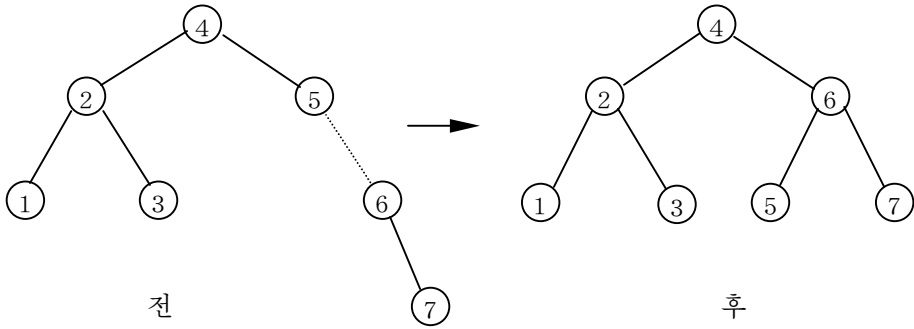



그림 4-22. 4번째 경우를 조정하는 단일회전

파괴되기때문이다. 그러므로 뿌리와 그의 왼쪽 자식사이에서 단일회전을 실행하여 그 문제를 조종한다. 여기에 조종하기전과 조종한 다음의 상태를 보여 준다.

점선은 회전의 대상인 두개의 매듭을 련결한다. 다음에 4를 삽입하는데 이것은 문제로 되지 않지만 5의 삽입은 단회전으로 조정된 매듭 3에서 AVL성질을 파괴시킨다. 더우기 회전에 의해 발생되는 국부적인 변화외에 프로그람작성자는 이 변화가 나무의 나머지에 미치는 정보를 알고 있어야 한다. 여기서 이것은 2의 오른쪽 자식으로 3대신에 4와 련결되도록 재설정되여야 한다는것을 의미한다. 그 작업은 잃어 버리기 쉬우며 나무를 파괴한다(4는 접근불가능하다.).

　다음에 6을 삽입한다. 이것은 뿌리에서 균형파괴를 발생시키는데 그것은 뿌리의 왼쪽 부분나무의 높이는 0이고 오른쪽 부분나무의 높이는 2이기때문이다. 그러므로 2와 4사이에 있는 뿌리에서 단일회전을 실행한다.

회전은 2를 4의 자식으로, 4의 초기 왼쪽 부분나무를 2의 새로운 오른쪽 부분나무로 만드는것으로 처리된다. 이 부분나무에서 모든 항목은 2와 4사이에 전개되여야 한다. 따라서 이 변환은 의미를 가지게 된다. 삽입되는 다음의 항목은 7인데 그것은 또 다른 회전을 발생한다.

2. 2중회전

우에서 작성된 알고리듬에는 한가지 문제점이 있다. 즉 그림 4-23에서 보여 주는것처럼 그 알고리듬은 두번째나 세번째 경우에 대하여서는 처리하지 못한다. 그 리유는 부분나무 *Y*가 너무 깊어서 단일회전으로서는 그의 깊이를 작아 지게 하지 못하기때문이다. 이 문제를 해결하는 2중회전(double rotation)을 그림 4-24에 보여 주었다.

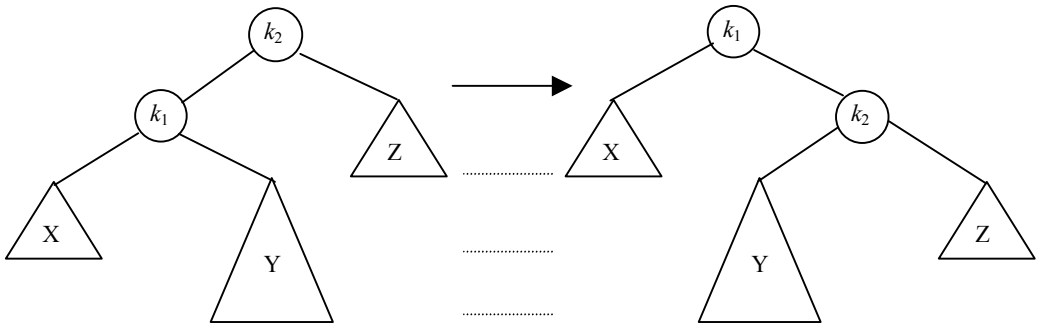



그림 4-23. 두번째 경우를 조정하는 단일회전의 실례

그림 4-23에서 부분나무 *Y*가 그안에 삽입된 어떤 항목을 가지고 있다는 사실은 부분나무 *Y*가 비지 않았다는것을 말해 준다. 따라서 그것은 하나의 뿌리와 2개의 부분나무들을 가진다고 가정할수 있다. 그림에서 보는것처럼 정확히 나무 *B*나 *C*중에서 하나는 *D*보다 2개 준위 더 깊지만(만일 모두 비지 않았으면) 그것이 어느것인가는 알수 없다. 그러나 그것은 문제로 되지 않는다. 즉 그림 4-24에서 *B*와 *C*중에서 하나는 *D*보다 1½준위만큼 낮은곳에 배치된다.

균형을 보장하기 위하여서는 k_3을 뿌리매듭으로 할수 없으며 k_3과 k_1사이의 회전은

그림 4-23에서 보여 준것처럼 균형을 조정하지 못한다. 따라서 유일한 방안은 k_2를 새로운 뿌리로서 배치하는것이다. 이것은 k_1을 k_2의 왼쪽 자식에, 그리고 k_3을 k_2의 오른쪽 자식에 련결하며 4개 부분나무들에 대한 위치를 완전히 결정하게 한다. 그 결과로 나무가 AVL성질을 만족시킨다는것은 쉽게 알수 있으며 단일회전에 대한 경우와 마찬가지로 삽입하기전에 가졌던 높이를 계속 유지한다. 따라서 전체적인 균형조정과 높이갱신이 완전히 담보된다. 그림 4-25는 세번째 경우가 2중회전에 의해서 조정될수 있다는것을 보여준다. 두 경우에 균형조정은 α의 자식과 α의 손자사이의 회전, 그다음 α와 그의 새자식사이의 회전에 의해서 처리된다.

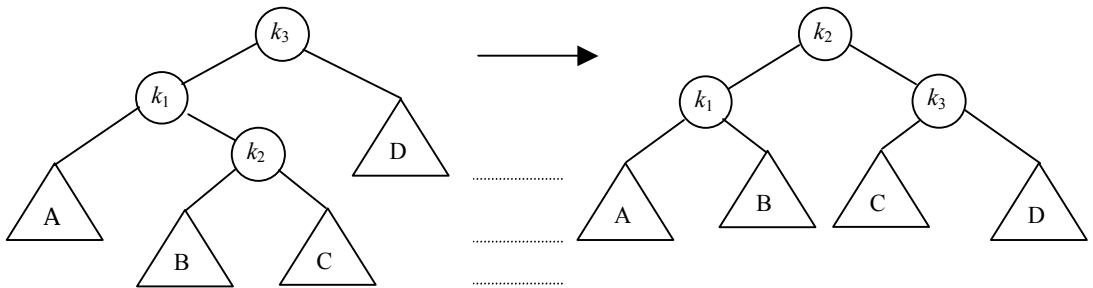



그림 4-24. 두번째 경우를 조정하기 위한 왼쪽-오른쪽 2중회전

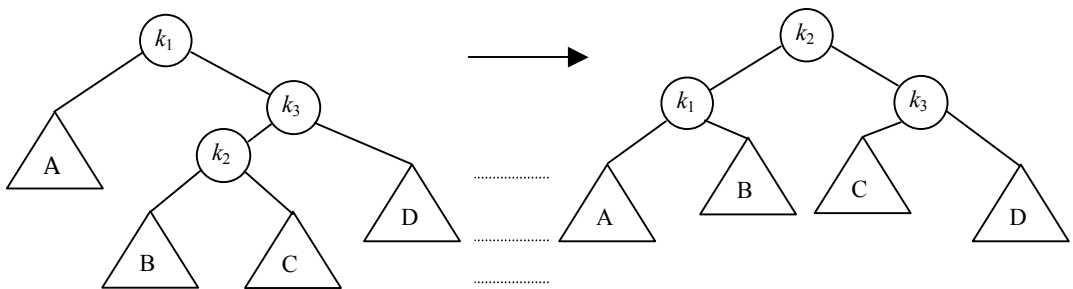
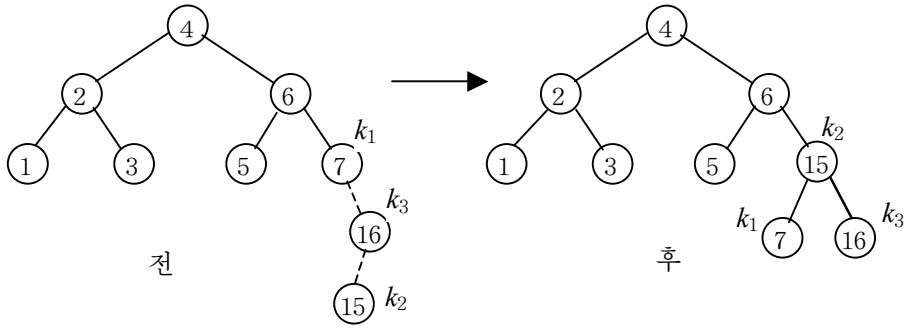
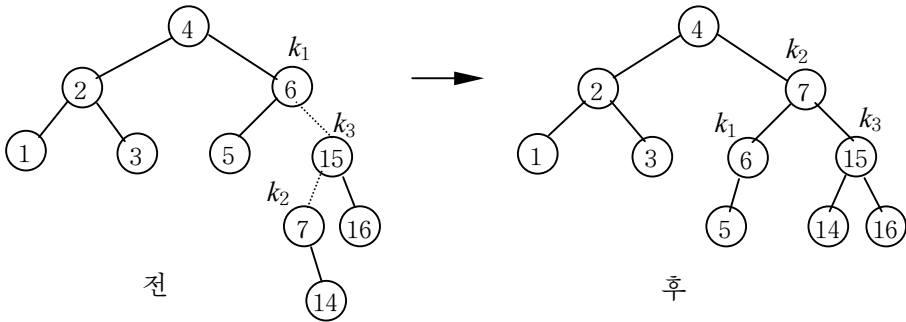
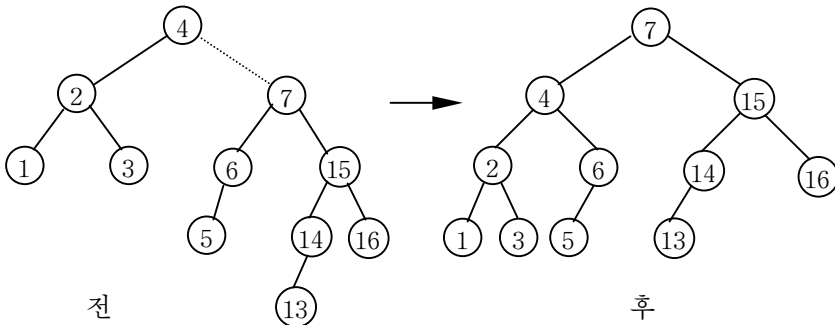
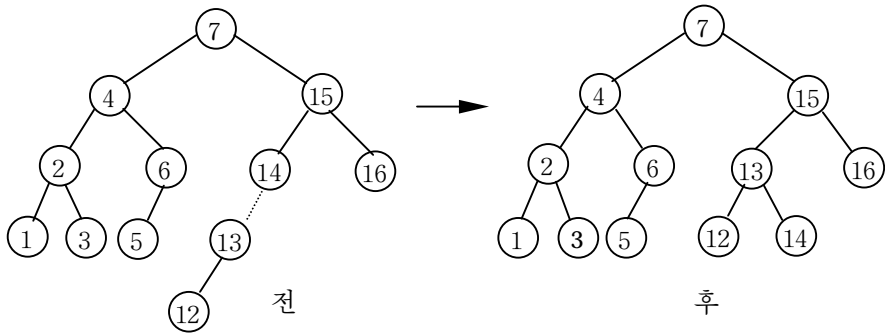
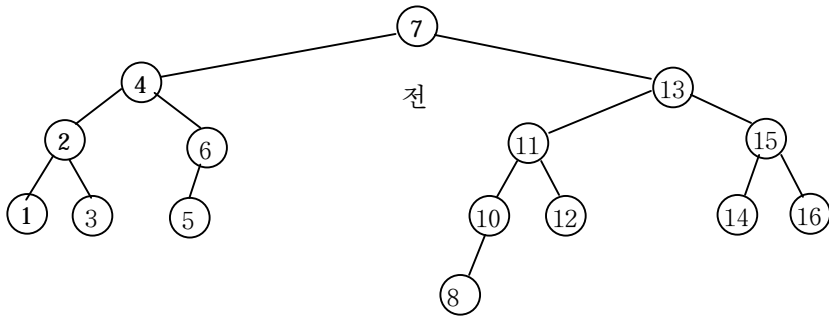
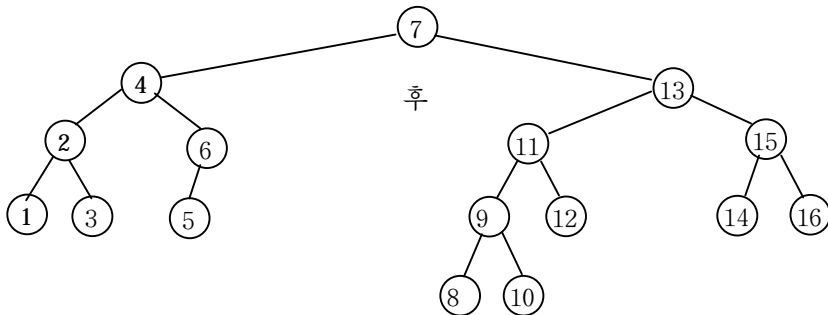



그림 4-25. 세번째 경우를 조정하기 위한 오른쪽-왼쪽 2중회전

우의 실례에 대하여 10부터 16까지 거꿀순서로 련속 삽입하고 다음 8, 9를 삽입하자. 16을 삽입하는것은 그것이 균형성질을 파괴시키지 않으므로 다른것이 없지만 15를 삽입할 때는 매듭 7에서 균형이 파괴된다. 이것은 세번째의 경우에 속하는데 오른쪽-왼쪽 2중회전으로 해결될수 있다. 우의 실례에서 오른쪽-왼쪽 2중회전은 7과 16,15를 포함하게 된다. 이 경우에 k_1은 항목 7을 가진 매듭이고 k_3은 항목 16을 가진 매듭이며 k_2는 항목 15를 가진 매듭이다. 부분나무 A, B, C, D들은 비여 있다.

다음에 14를 삽입하는데 역시 2중회전을 요구한다. 여기서 2중회전은 6과 15, 17을 포함하는 오른쪽-왼쪽 2중회전으로 처리된다. 이 경우에 k_1은 항목 6을 가진 매듭이고 k_2는 항목 7을 가진 매듭이며 k_3은 항목 15를 가진 매듭이다. 부분나무 A는 항목 5를 가진 매듭에 고정된 나무이며 부분나무 B는 처음에 항목 7을 가진 매듭의 왼쪽 자식이였던 빈 부분나무이며 부분나무 C는 항목 14를 가진 매듭에 고정된 나무이다. 또한 부분나무 D는 항목 16을 가진 매듭을 뿌리로 하는 나무이다.

이제 13이 삽입되면 뿌리에서 균형이 파괴된다. 13이 4와 7사이에 없으므로 단일회전으로서 균형을 조정할수 있다.

12의 삽입도 역시 단일회전을 요구한다.

11을 삽입하면 단일회전을 실행하여야 하는데 이러한 처리는 다음에 10을 삽입할 때에도 요구된다. 다음에는 회전이 없이 8을 삽입하여 거의 완전한 균형나무를 얻는다.

마지막으로 9를 삽입하여 2중회전의 대칭경우를 고찰할수 있다. 이때 9는 10의 부분나무에 삽입되므로 균형을 파괴한다. 9가 10과 8(이것은 9에 가는 경로상에 있는 10의 자식이다.)사이에 있으므로 2중회전이 실행되여야 하며 그것은 다음과 같은 나무를 만들어 낸다.

지금까지의 과정을 요약하자. 여러가지 경우를 제외하면 상세한 프로그람작성은 아주 간단하다. AVL나무 T에 항목 X를 가진 새로운 매듭을 삽입하기 위하여 T의 적당한 부분나무에 X를 재귀적으로 삽입한다(이것을 T_{LR}라고 하자.). 만일 T_{LR}의 높이가 변경되지 않으면 처리를 끝낸다. 그러나 T에 불균형높이가 나타나면 X와 T와 T_{LR}항목들에 기

초하여 적당히 단일회전 또는 2중회전을 실행하여 높이들을 수정하고(우에서의 나머지의 나무로부터 련결되는) 처리를 완료한다. 단일회전은 항상 성공하므로 세밀하게 코드화된 비재귀적인 처리는 일반적으로 재귀적인 처리보다 훨씬 더 빠르다. 그러나 비재귀적인 처리는 정확한 코드작성이 매우 어려우므로 많은 프로그람작성자들은 AVL나무를 재귀적으로 실현한다.

다른 또 하나의 효과적인 문제는 높이정보에 영향을 주는것이다. 실제로 요구되는것은 모든 매듭들에 대하여 그의 부분나무들의 높이차가 작아 지도록 하는것인데 그것은 2개의 비트(+1, 0, -1을 나타내기 위하여)로써 표현할수 있다. 이것은 명백하지 못한 어떤 결과로 발생하는 균형결수들에 대한 반복적인 계산을 피하게 한다. 결과적인 코드는 높이가 매개 매듭에 보관될 때보다 좀 더 복잡하다. 만일 재귀적인 루틴을 리용하면 속도는 중요하게 고려하지 않아도 된다. 이 경우에 균형결수들을 유지하여 얻어 지는 우점은 그리 명백치 않으나 상대적으로 간단하다. 더우기 대부분의 기계들은 어떤 경우에도 정보를 적어도 8bit로 취급하기때문에 리용되는 공간량에서는 차이가 있을수 없다. 어떤 8bit기호는 127까지의 절대적인 높이를 보관할수 있다. 나무가 균형으로 되면 이것이면 충분하다(련습문제를 보시오.).

이 모든것을 가지고 AVL루틴들을 작성할수 있다. 여기서 일부 코드를 고찰하는데 그 나머지는 직결루틴들이다. 먼저 AvlNode클라스가 필요하다. 이것은 프로그람 4-15에서 보여 주었다. 또한 매듭의 높이를 되돌리는 고속함수가 필요하다. 이 함수는 어떤 NULL지적자에 대한 시끄러운 경우를 조종하는데 필요하다. 이것은 프로그람 4-16에서 보여 준다. 프로그람 4-17에서 알수 있는것처럼 기본삽입루틴은 대부분 함수호출들로 구성되므로 서술하기가 쉽다.

```
template <c1ass Comparab1e>
class AvlTree;

template <c1ass Comparab1e>
class AvlNode
{
        Comparable element;
        AvlNode    *left;
        AvlNode    *right;
        int             height;
        AvlNode( const Comparable & theElement, AvINode *lt,
                                                        AvlNode *rt, int h = 0 )
         : e1ement( theElement ), 1eft(lt ), right( rt ), height( h ) { };
        friend class AvlTree<Comparab1e>;
};
```

프로그람 4-15. AVL나무에 대한 매듭선언

```
/**
 * Return the height of node t, or -1, if NULL.
 */
template <c1ass Comparab1e>
int AvlTree<Comparab1e>::height( AvlNode<Comparab1e> *t )
const
{
      return t == NULL ? -1 : t->height;
}
```

프로그램 4-16. AVL매듭의 높이를 계산하는 함수

```
/**
 * Internal method to insert into a subtree.
 * x is the item to insert.
 * t is the node that roots the tree.
 */
template <c1ass Comparab1e>
void AvlTree<Comparab1e>::insert( const Comparable & x,
                                  AvlNode<Comparab1e> * & t ) const
{
    if( t == NULL )
       t = new AvlNode<Comparab1e>( x, NULL, NULL );
    else if( x < t->e1ement )
    {
       insert( x, t->1eft );
       if( height( t->1eft ) - height( t->right ) == 2 )
          if( x < t->1eft->e1ement )
             rotateWithLeftChild( t );
          else
             doub1eWithLeftChild( t );
    }
    else if( t->e1ement < x )
    {
       insert( x, t->right );
       if( height( t->right ) - height( t->left ) == 2 )
          if( t->right->e1ement < x )
             rotateWithRightChild( t );
          else
             doubleWithRightChild( t );
    }
    else
       ;  // Duplicate; do nothing
    t->height = max( height( t->left ), height( t->right ) ) + 1;
}
```

프로그램 4-17. AVL나무에로의 삽입

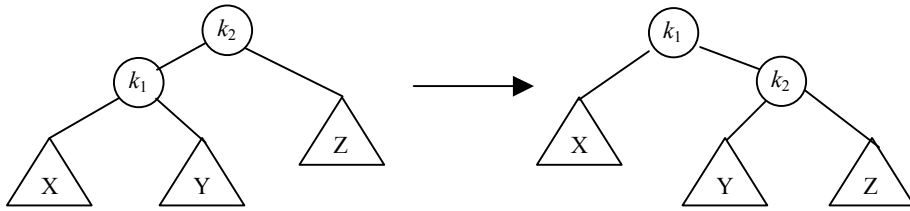



그림 4-26. 단일회전

rotateWithLeftChild는 그림 4-26에 있는 나무들에 대하여 왼쪽에 있는 나무를 오른쪽에 있는 나무로 변환하고 새로운 뿌리에 대한 지적자를 되돌린다. rotateWithLeftChild는 대칭적이다. 그 코드는 프로그람 4-18에서 보여 준다.

```
/**
  * Rotate binary tree node with left child.
  * For AVL trees, this is a single rotation for case 1,
  * Update heights, then set new root.
  */
template <c1ass Comparab1e>
void AvlTree<Comparab1e>::
rotateWithLeftChild( AvlNode<Comparab1e> * & k2 ) const
{
        AvlNode<Comparab1e> *kl = k2->left;
        k2->1eft = kl->right;
        kl->right = k2;
        k2->height = max( height( k2->left ), height( k2->right ) ) + 1;
        kl->height = max( height( kl->left ), k2->height ) + 1;
        k2 = kl;
}
```

프로그람 4-18. 단일회전을 실행하는 루틴

마지막으로 그림 4-27에서 보여 준 2중회전을 실행하는 코드는 프로그람 4-19에서 보여 준다. AVL나무에서 삭제는 삽입보다 좀 더 복잡한데 그것을 련습문제로 남겨 둔다. 지연삭제는 대체로 삭제들이 상대적으로 드물게 필요할 때 가장 좋은 방법으로 된다.

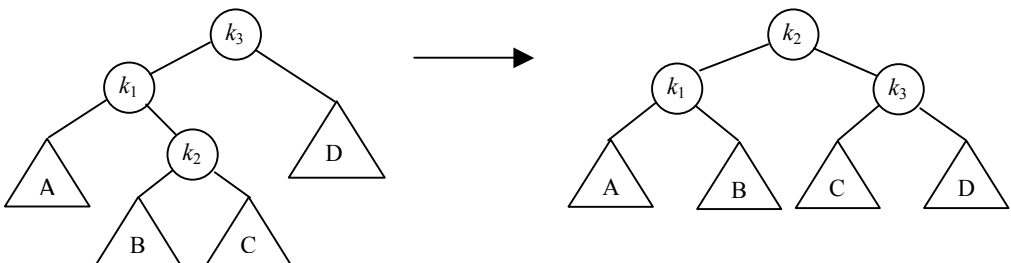



그림 4-27. 2 중회전

```
/**
  * Double rotate binary tree node: first left child.
  * with its right child; then node k3 with new left child.
  * For AVL trees, this is a double rotation for case 2.
  * Update heights, then set new root.
  */
template <c1ass Comparab1e>
void AvlTree<Comparab1e>::
doub1eWithLeftChild( AvlNode<Comparab1e> * & k3 ) const
{
    rotateWithRightChild( k3->left );
    rotateWithLeftChild( k3 );
}
```

프로그람 4-19. 2중회전을 실행하는 루틴

제5절. 펼친나무

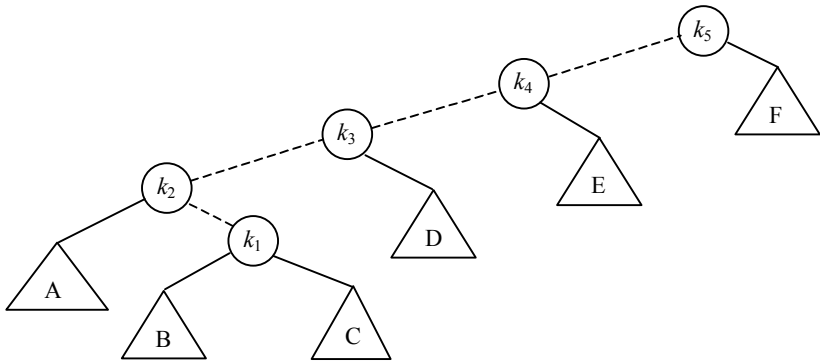
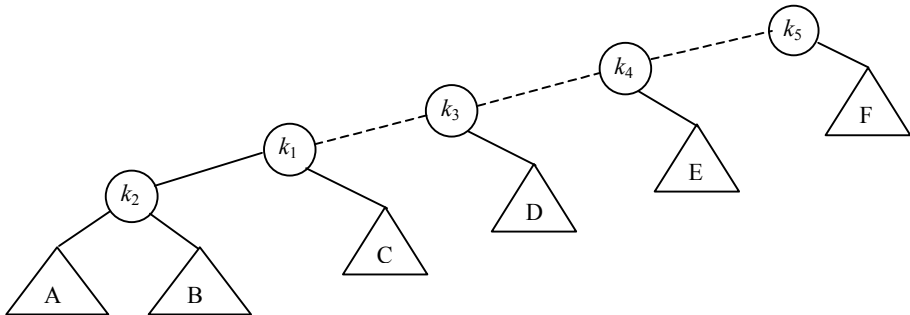
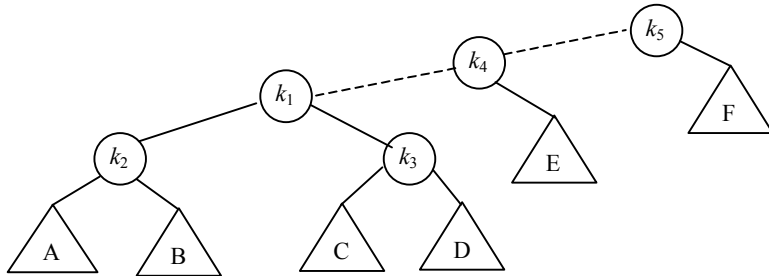
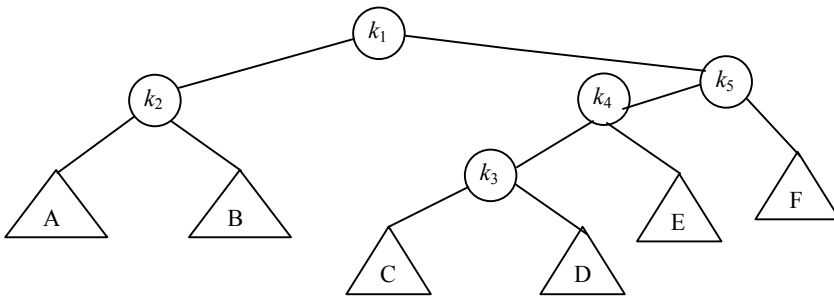
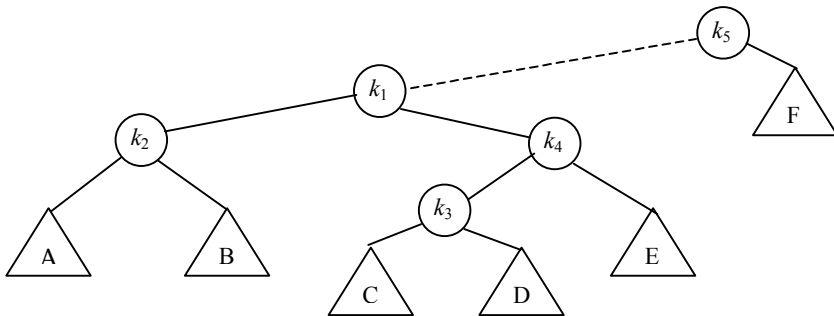
이제부터 펼친나무(splay tree)라고 하는 상대적으로 간단한 자료구조를 고찰하자. 펼친나무는 빈나무로부터 시작하여 임의의 M개의 련속적인 나무연산들이 많아서 $O(M\log N)$ 이하의 시간에 처리되도록 만들어 진 나무이다. 이 나무는 어떤 하나의 연산이 $O(N)$시간에 수행될 가능성을 배제하지 않으며 따라서 그 한계가 매 연산당 최악의 경우의 한계 $O(\log N)$보다는 크지 않다고 해도 근본적인 효과는 같다. 즉 나쁜 입력렬은 없다. 일반적으로 M개 연산들의 서렬이 최악의 경우에 $O(Mf(N))$의 실행시간을 가지면 실행시간은 $O(f(N))$으로 된다. 따라서 펼친나무는 매개 연산당 $O(\log N)$의 비용을 가진다. 연산들의 긴 서렬에서 일부는 더 큰 값을 가지며 또 어떤 일부는 더 작은 값을 가진다.

펼친나무는 2진탐색나무에서 매 연산당 최악의 경우의 시간 $O(N)$이 결코 나쁜것이 아니라는 사실에 기초하고 있는데 그것은 상대적으로 드물게 발생한다. 어떤 하나의 접근은 그것이 $O(N)$을 가질 때에도 여전히 매우 빠르다. 2진탐색나무에서 문제는 나쁜 접근들 전체 서렬에 대하여 그것이 가능하며 흔치 않다는것이다. 그때 루적실행시간은 주목할만한것이다. 최악의 경우 $O(N)$시간을 가지는 탐색나무자료구조는 임의의 M개의 련속적인 연산들에 대하여 많아서 $O(M\log N)$을 담보하는것으로서 아주 효과적인데 그것은 거기에 부당한 서렬이 없기때문이다.

만일 어떤 개별적인 연산이 최악의 경우에 $O(N)$의 시간한계를 가지도록 작성되고 사용자는 여전히 $O(\log N)$의 시간한계를 바란다면 어떤 매듭이 접근될 때마다 그 매듭이 이동되여야 한다. 한편 깊은 매듭을 찾기만 하면 그에 대한 find연산들을 유지한다. 매듭이 위치를 변경하지 않고 매 접근이 $O(N)$값을 가지면 M개의 접근서렬은 $O(M\cdot N)$값을 가진다.

펼친나무의 기본사상은 어떤 매듭이 접근된 다음에 AVL나무의 회전들에 의해 그 매듭을 뿌리에 넣는것이다. 매듭이 깊으면 그 경로상에는 상대적으로 깊은 매듭들이 많으며 재구축으로써 이 모든 매듭들에 앞으로 더 쉽게 접근할수 있다. 만일 매듭이 지나치게 깊으면 나무를 재구축하여 어느 정도 균형을 조정한다. 이 산법은 리론적으로 좋은 시간한계를 주며 더우기 많은 응용들에서 매듭이 접근될 때 그 매듭이 불원간 다시 접근되여야 하기때문에 실천적으로 유익하다. 이에 대한 연구들은 예상보다 훨씬 더 많이 진행되였다. 또한 펼친나무들은 높이의 유지나 균형정보들을 요구하지 않는다. 따라서 어느정도 공간을 절약하고 코드를 간단하게 한다(특히 구체적인 실현들이 서술될 때).

1. 간단한 개념

앞에서 설명된 재구축을 실현하는 한가지 방법은 단일회전들을 거꾸로 실행하는것이다. 이것은 접근경로상에 있는 모든 매듭들이 자기의 부모와 함께 회전한다는것을 의미한다. 실례로 다음의 나무에서 k_1에 접근(find)한 다음에 일어 나는 과정을 고찰해 보자.

접근경로는 점선이다. 먼저 k_1과 그의 부모사이에서 단일회전을 수행한다. 그 결과는 다음의 나무와 같다.

그다음 k_1과 k_3사이에서 회전하여 다음의 나무를 얻는다.

그다음에 뿌리매듭에 도달할 때까지 두번의 회전을 더 실행한다.

이 회전들은 뿌리에로 가는 모든 경로들에 k_1을 배치하게 한다. 그러므로 k_1에 대한 앞으로의 접근은 쉽게 처리된다. 그러나 다른 매듭(k_3)이 초기의 k_1만큼 깊은 곳에 배치된다. 그 매듭에 대한 접근은 또 다른 매듭을 깊어 지게 한다. 이 방법이 k_1에 대한 앞으로의 접근을 더 쉽게 한다고 해도 본래의 접근경로상의 다른 매듭들의 처지는 대폭 개선하지 못한다. 이 방법을 리용하면 M개 연산들의 서렬이 $\Omega(M \cdot N)$시간을 요구하므로 이 방법은 그다지 좋은것이 아니다. 이것을 보여 주는 가장 간단한 방법은 초기의 빈나무에 열쇠 $1, 2, 3, 4, \cdots, N$을 삽입하여 만들어 진 나무를 고찰하는것이다(이 실례를 파악하시오.). 이것은 왼쪽 자식들만으로 구성된 나무를 만든다. 이 나무를 구축하는데 걸리는 시간은 총체적으로 $O(N)$이므로 이 나무는 그다지 나쁘지는 않다. 그런데 결합은 열쇠 1

을 가진 매듭에 접근할 때 N-1의 단위시간을 가지는것이다. 회전들이 끝난 다음에 열쇠 2를 가진 매듭에 대한 접근은 N-1시간단위를 가진다.

모든 열쇠들에 차례로 접근하는데 걸리는 총 시간은 $\sum_{i=1}^{N-1} i = \Omega(N^2)$ 이다. 그것들이 다 접근되면 나무는 초기상태에로 되돌아 가며 절차를 반복할수 있다.

2. 펼치기

펼치기는 회전들이 실현되는 방법이 좀 더 선택적이라는것을 제외하고 앞에서 고찰한 회전개념과 류사하다. 여전히 접근경로를 따라서 아래에서 우로 회전하자. X를 접근경로상에 있는 회전하게 될 매듭(뿌리매듭이 아님)이라고 하자. 만일 X의 부모가 나무의 뿌리이면 X는 뿌리와 함께 회전한다. 이것은 접근경로에 따르는 마지막회전이다. 한편 X가 부모(P)와 조부모(G)를 가지면 대칭적인 두가지 경우가 있다. 첫번째 경우는 **왼쪽-오른쪽**(zig-zag case)경우이다(그림 4-28). 여기서 X는 오른쪽 자식이고 P는 왼쪽 자식이다(또는 반대). 이 경우에 AVL나무에서와 같이 2중회전을 실행한다. 그렇지 않은 경우는 **왼쪽-왼쪽**(zig-zig)경우인데 여기서 X와 P는 둘다 왼쪽 자식들이다(또는 그 대칭경우로서 둘다 오른쪽 자식들이다). 이 경우에 그림 4-29에서 왼쪽에 있는 나무를 오른쪽에 있는 나무로 전환한다.

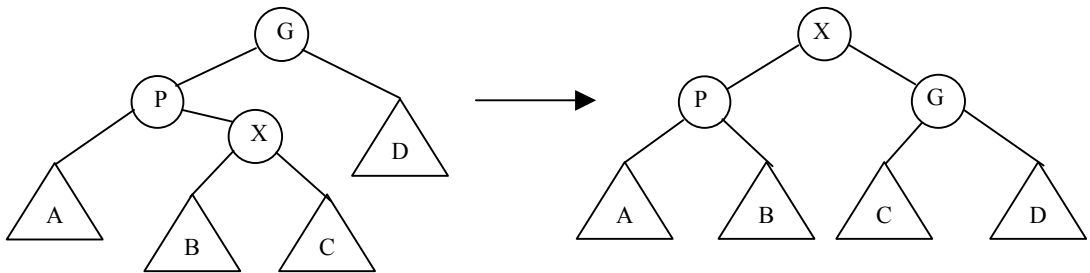



그림 4-28. 왼쪽-오른쪽

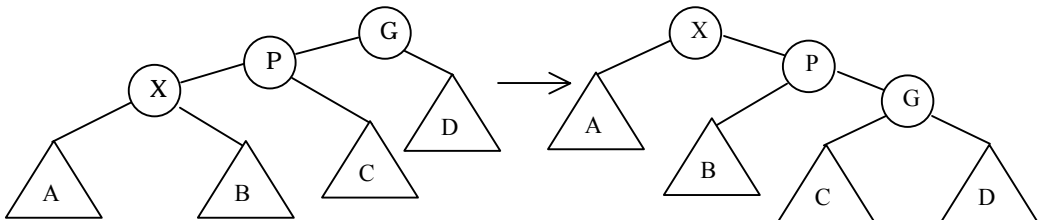
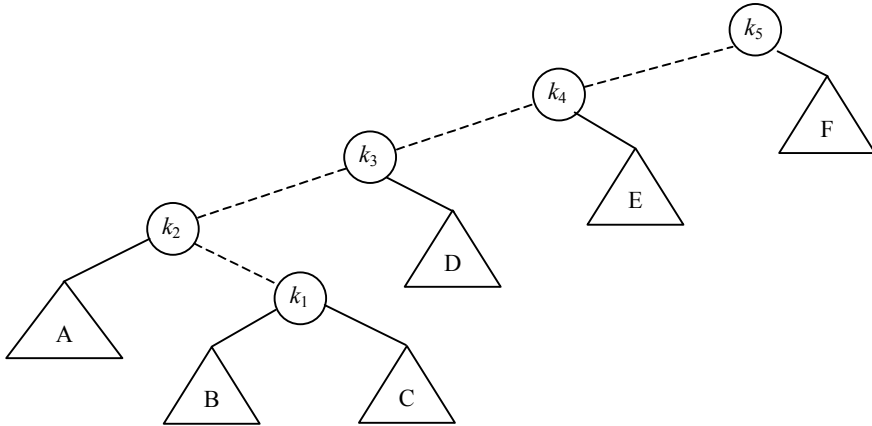
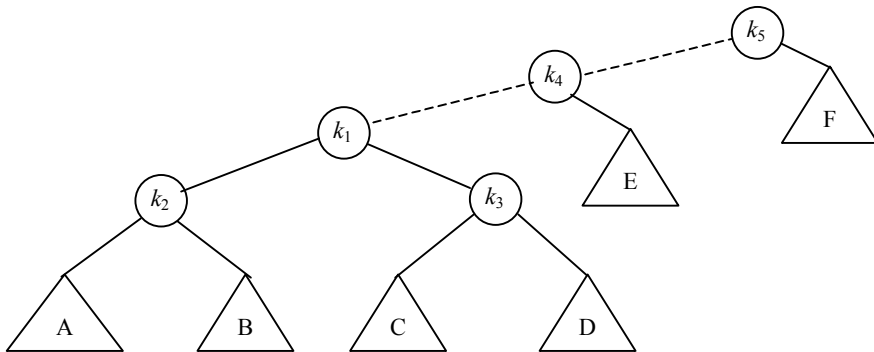
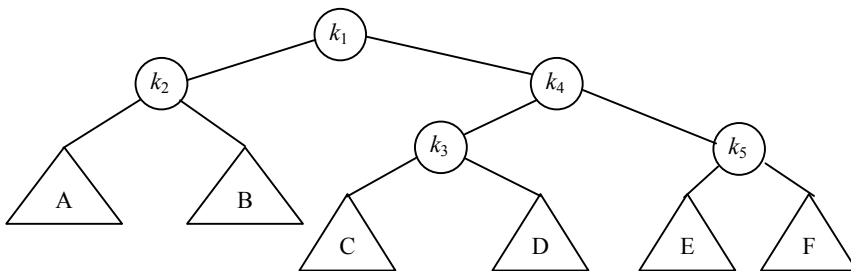



그림 4-29. 왼쪽-왼쪽

실례로 k_1에 대한 find를 가지고 마지막실례의 나무를 고찰하자.

첫번째 펼침걸음은 k_1에 있으며 정확히 왼쪽-오른쪽이다. 그래서 k_1과 k_2, k_3을 리용하여 표준AVL의 2중회전을 수행한다. 그 결과는 다음과 같다.

k_1에서 다음 펼침걸음은 왼쪽-왼쪽이다. 그래서 k_1과 k_4, k_5를 가지고 왼쪽-왼쪽회전을 수행하여 마지막나무를 얻는다.

비록 작은 실례에서는 잘 알리지 않지만 펼치기는 접근된 매듭을 뿌리에로 이동할

뿐아니라 접근경로상의 거의 모든 매듭들의 깊이를 약 절반으로 줄이는 효과를 가진다 (일부 얕은 매듭들은 많아서 2개 준위만큼 내려 간다.).

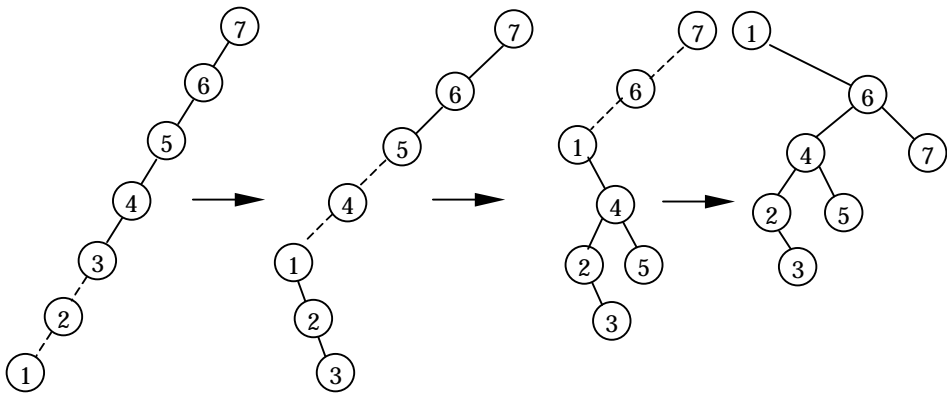



그림 4-30. 매듭 1에서 펼침처리의 결과

펼치기가 회전을 간단하게 하는 차이를 알기 위하여 처음의 빈나무에 항목 $1, 2, 3, \cdots, N$을 삽입하는 효과를 다시 고찰하자. 이것은 앞에서와 같이 총 $O(N)$의 시간을 가지며 간단한 회전으로 같은 나무를 만든다. 그림 4-30은 항목 1을 가진 매듭에서 펼침처리의 결과를 보여 준다. 그 차이는 N-1시간단위를 가지는 항목 1을 가진 매듭에 접근한 다음에 항목 2를 가진 매듭에 대한 접근은 N-2시간단위가 아니라 약 $N/2$시간단위를 가지게 된다는것이다. 펼침처리전과 같은 깊이를 가지는 매듭들은 하나도 없다.

항목 2를 가진 매듭에 대한 접근은 그 매듭을 뿌리의 $N/4$이내에 옮겨 오며 이것은 깊이가 약 $\log N$으로 될 때까지 반복된다(N=7을 가진 실례는 너무 작아서 그 효과를 잘 알수 없다.). 그림 4-31～4-39는 처음에 왼쪽 자식만을 포함하는 32개 매듭으로 이루어진 나무에서 1부터 9까지의 항목들을 접근한 결과를 보여 준다. 따라서 간단한 회전방법으로 널리 쓰이는 펼친나무에서 나쁜 특성은 없다(실제로 이것은 아주 좋은 경우이다. 좀 더 복잡한 증명은 이 실례에서 N개의 접근들이 총 $O(N)$시간을 가진다는것을 보여 준다.).

이 그림들은 펼친나무들의 기초적이며 중요한 성질들을 뚜렷이 보여 준다. 접근경로들이 길어 질 때 표준탐색시간보다 더 길어 지게 하면서 회전들은 앞으로의 연산들에 더 유리해 지게 한다. 접근들이 간단히 얻어 지면 그 회전들은 좋지 않으며 질이 나쁠수 있다. 극단한 경우는 초기나무가 삽입들에 의해서 만들어 질 때이다. 삽입들은 모두 질이 나쁜 초기나무를 만드는 상수시간연산들이다. 그 시점에서 나무의 질이 매우 나쁘지만 전처리를 실행하여 전체 실행시간을 작게 하였다. 그때 2개의 접근이 균형나무에 남아 있지만 그의 비용은 절약된 시간을 얼마간 돌려 가지면 된다. 제11장에서 보게 될 중요한 정리는 매 연산당 시간이 $O(\log N)$이하로는 떨어 지지 않는다는것이다. 즉 혹간 질이

나쁜 연산들이 있다고 해도 언제나 처리중에 있게 된다.

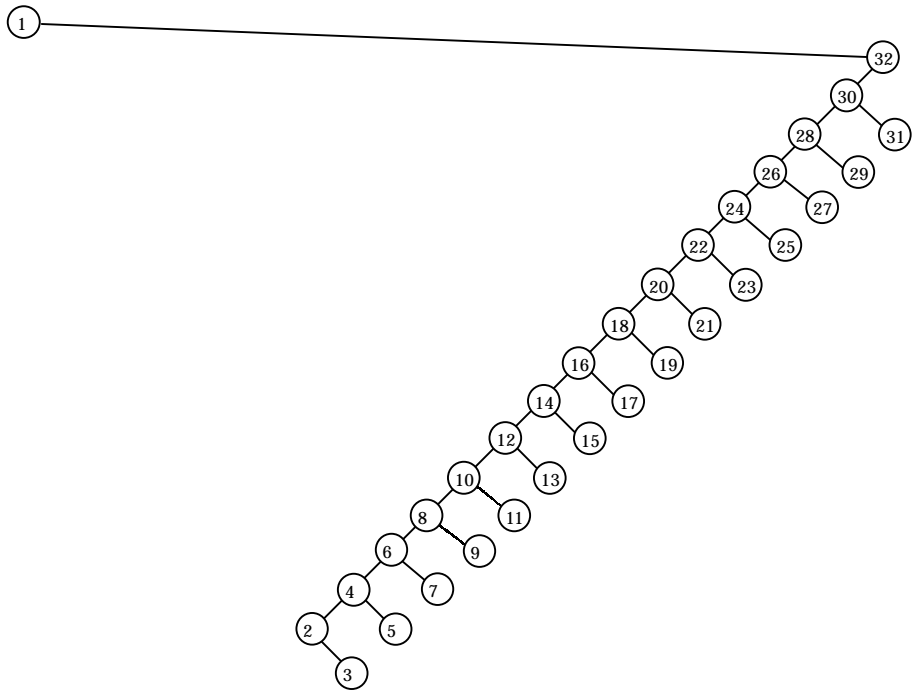



그림 4-31. 모두 왼쪽 자식만을 가지는 나무를 매듭 1에서 펼침처리한 결과

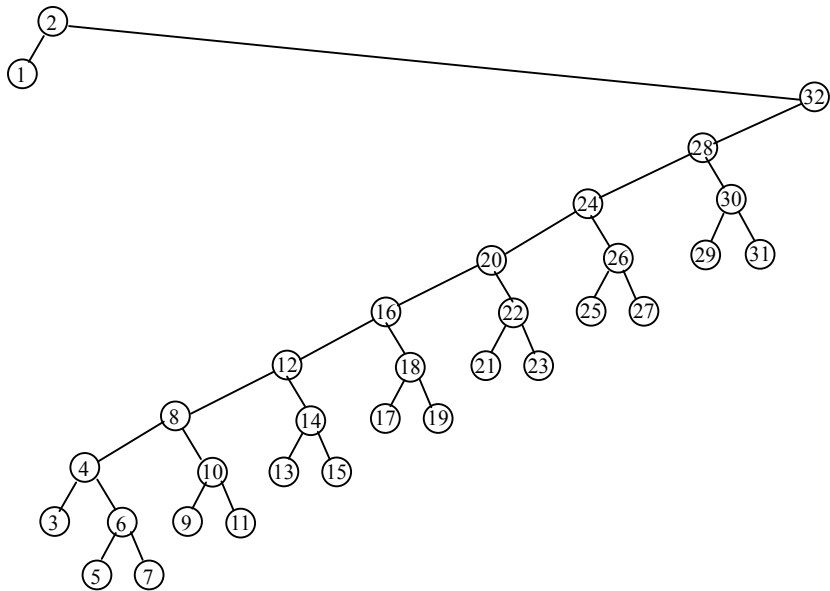



그림 4-32. 매듭 2에서 전단계의 나무를 펼침처리한 결과

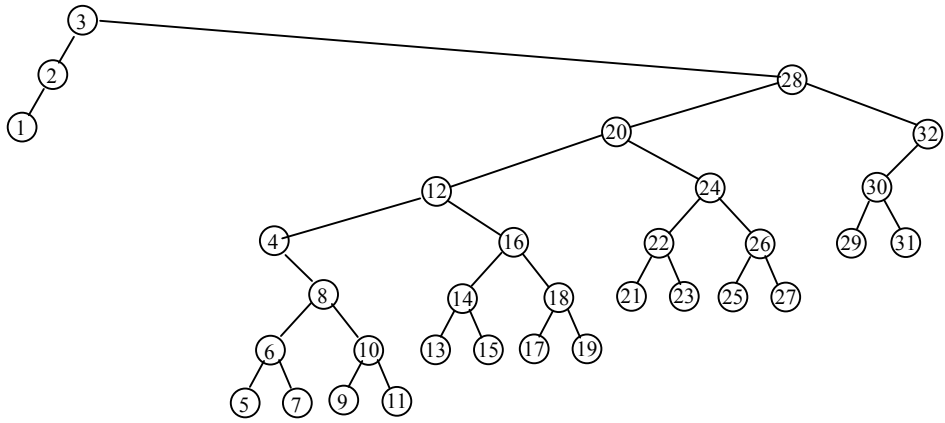



그림 4-33. 매듭 3에서 전단계의 나무를 펼침처리한 결과

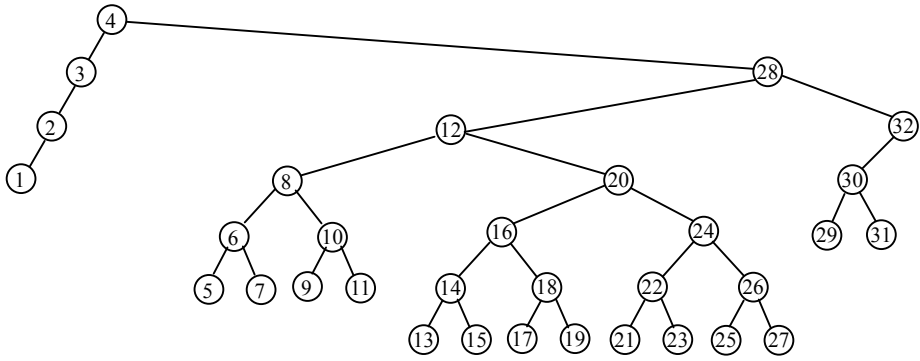



그림 4-34. 매듭 4에서 전단계의 나무를 펼침처리한 결과

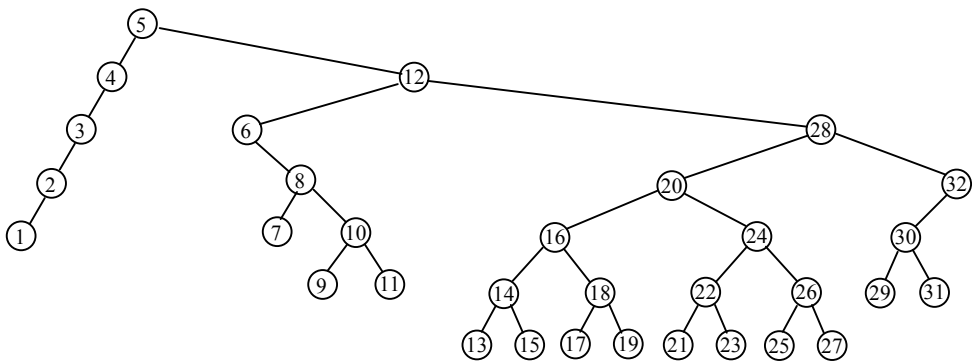



그림 4-35. 매듭 5에서 전단계의 나무를 펼침처리한 결과

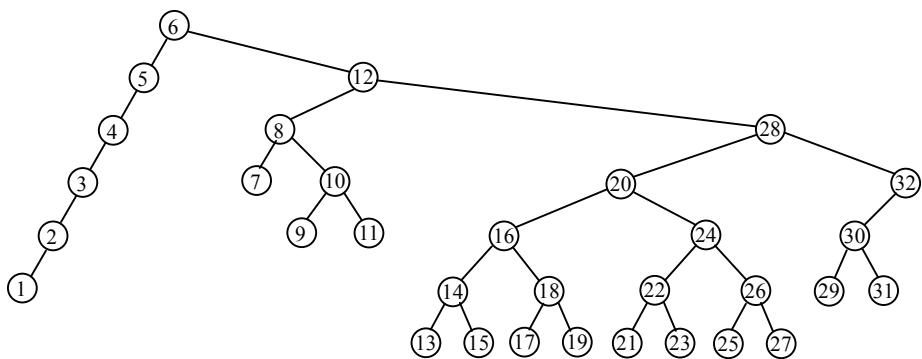



그림 4-36. 매듭 6에서 전단계의 나무를 펼침처리한 결과

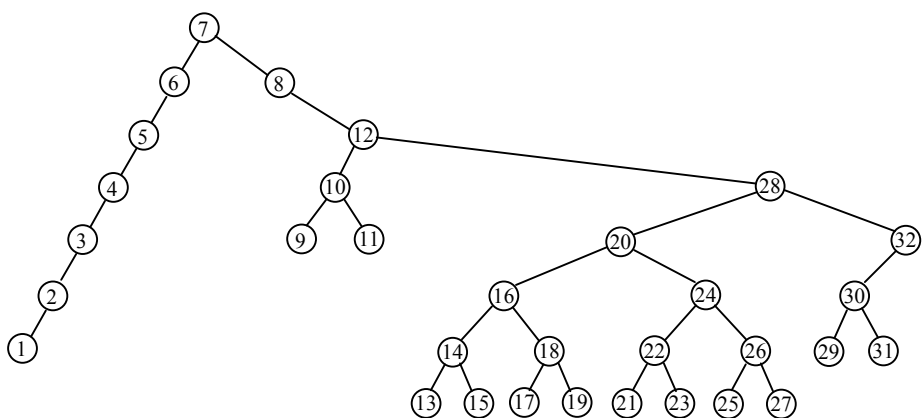



그림 4-37. 매듭 7에서 전단계의 나무를 펼침처리한 결과

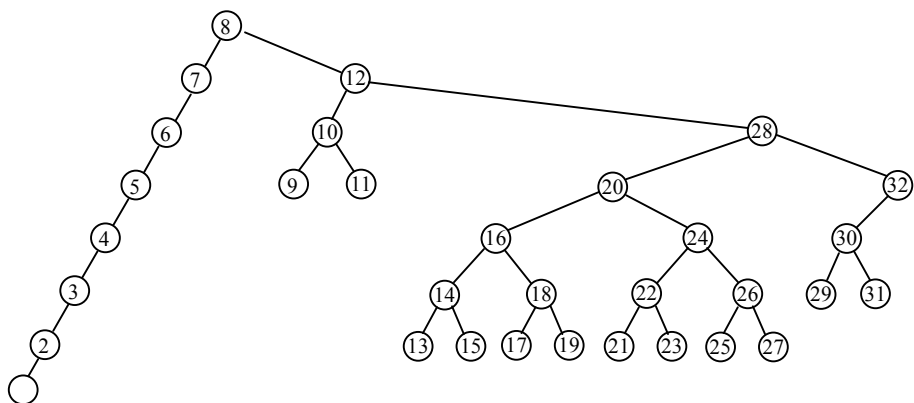



그림 4-38. 매듭 8에서 전단계의 나무를 펼침처리한 결과

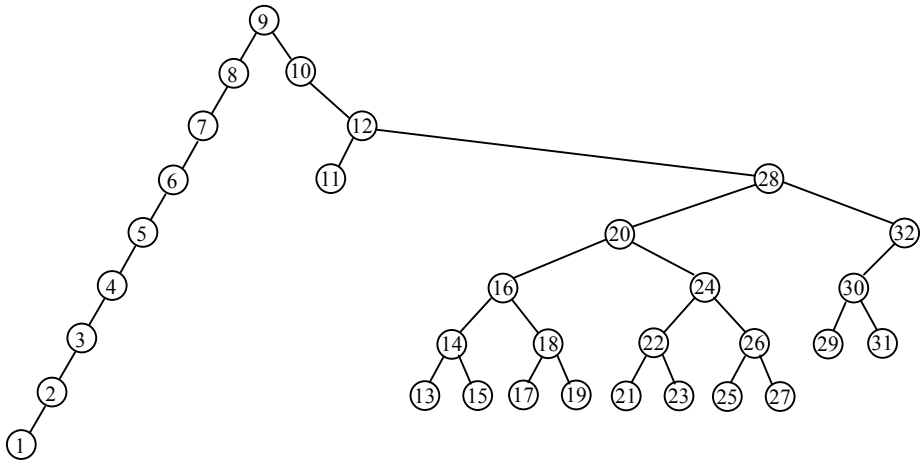



그림 4-39. 매듭 9에서 전단계의 나무를 펼침처리한 결과

삭제하여야 할 매듭에 접근하여 삭제를 실행할수 있다. 먼저 삭제할 매듭을 뿌리에 놓는다. 그것이 삭제되면 두개의 부분나무 T_L과 T_R를 얻는다(왼쪽과 오른쪽에서). 만일 T_L에서 가장 큰 요소를 찾으면(그것은 쉽다.) 그 요소는 T_L의 뿌리에로 회전되며 T_L은 그때 오른쪽 자식을 가지지 않는 뿌리를 가지게 된다. T_R를 오른쪽 자식으로 만들면 삭제가 완료된다.

제6절. 나무의 순회

2진탐색나무에서는 순서정보가 있기때문에 모든 항목들을 정렬된 순서로 간단히 표시할수 있다. 프로그람 4-20의 재귀함수는 그에 대한 실제적인 처리를 수행한다. 이 함수의 처리과정을 주의깊게 살펴 보자. 앞에서 고찰한것처럼 나무에 적용된 이러한 종류의 루틴을 **중뿌리순회**라고 한다(이것은 항목들을 순서적으로 표시하기때문에 리치에 맞는다). 중뿌리순회의 일반적인 방법은 왼쪽 부분나무를 먼저 처리하고 다음에 현재의 매듭을 처리하고 마지막으로 오른쪽 부분나무를 처리하는것이다. 이 알고리듬에서 흥미 있는것은 그것이 단순하다는것외에 전체 실행시간이 $O(N)$이라는것이다. 이것은 나무의 모든 매듭에서 상수적인 처리가 존재하기때문이다. 매개 매듭은 한번 방문되며 거기에서는 NULL에 대하여 검사하고 두개의 함수호출을 설정하고 출력지령을 수행한다. 매 연산당 작업내용이 고정적이고 N개의 매듭들이 있으므로 그 실행시간은 $O(N)$이다.

```
/**
* Print the tree contents in sorted order.
*/
```

```
template <c1ass Comparab1e>
class BinarySearchTee
{
 public:
        explicit Bi'narySearchTreeC const Comparable & notFound );
        B-inarySearchTree( const Bi'narySearchTree & rhs );
        -BinarySearchTreeC );

        const Comparable & findMin( ),const;
        const Comparable & f-indMax( ) const;
        const Comparable & f"ind( const Comparable & x ) const;
        bool isEmpty( ) const;
        void printTree( ) const;

        void makeEmptyC );
        void insert( const Comparable & x );
        void remove( const Comparable & x );

         const BinarySearchTree & operator=( const BinarySearchTree &
 rhs );

private:
        B-inaryNode<Comparab1e> "root;
        const Comparable ITEM_NOT_FOUND;
```

프로그람 4-20. 2진탐색나무를 차례로 출력하는 루틴

때때로 어떤 매듭을 처리하기전에 먼저 2개의 부분나무들을 처리하여야 할 때가 있다. 실례로 매듭의 높이를 계산하려면 먼저 부분나무의 높이를 알아야 한다. 프로그람 4-21의 코드는 이것을 계산한다. 특수한 경우들을 조사하는것은 좋은 사상이며(그리고 재귀가 포함되면 어려운) 그 루틴은 잎매듭의 높이를 정확히 0으로 선언한다. 이러한 일반적인 순회방법은 역시 앞에서 본것과 같은데 이것을 후뿌리순회라고 한다. 또한 매개 매듭에 대하여 상수적인 처리가 실행되기때문에 전체 실행시간은 O(N)이다.

다음으로 일반적인 순회방안은 선뿌리순회이다. 여기서 매듭은 자식들보다 먼저 처리된다. 실례로 매개 매듭에 그 매듭의 깊이에 따라 준위를 할당하려 할 때에 효과적으로 리용할수 있다.

이 모든 루틴들에서 일반적인 사상은 먼저 NULL인 경우를 조정하고 그다음에 나머지경우를 조정하는것이다. 이때 외부변수들이 부족되지 않도록 주의하여야 한다. 이 루틴들은 부분나무의 뿌리로 되는 매듭에 대한 지적자만을 보내고 그 어떤 여분의 변수를 선언하지도 보내지도 않는다. 코드를 압축할수록 사소한 오유들이 줄어 들게 된다. 많이 리용되지 않는 다른 순회(이것은 아직 고찰하지 않았다.)는 준위순서순회(level-order)이다. 준위순서순회에서 깊이가 d인 모든 매듭들은 깊이가 d+1인 매듭들보다 먼저 처리된

다. 준위순서순회는 재귀적으로 수행되지 않는다는 점에서 다른 순회산법들과 차이나며 재귀산법에서 적용된 탄창대신에 대기렬을 리용한다.

```
/**
 * Internal method to compute the height of a subtree rooted at t.
 * Assumes this function is a friend of BinaryNode.
 */
template <c1ass Comparab1e>
int height( BinaryNode<Comparab1e> *t )
{
    if( t == NULL )
        return -1;
    else
        return 1 + max( height( t->left ), height( t->right ) );
}
```

프로그람 4-21. 후뿌리순회를 리용하여 나무의 높이를 계산하는 루틴

제7절. B-나무

지금까지는 콤퓨터의 주기억기에 전체 자료구조를 보관할수 있다고 가정하였다. 그러나 주기억기에 넣을수 있는것보다 더 많은 자료구조를 가지고 있고 따라서 디스크에 그 자료구조를 넣어야 한다고 하자. 이러한 때 큰O모형은 더이상 리용할수 없으므로 규칙이 변경되여야 한다.

문제는 큰O분석에서는 모든 연산들이 동등하다고 가정한다는것이다. 그러나 이것은 디스크입출력이 포함되면 옳지 않다. 실례로 25-MIPS기계는 초당 2천500만개의 명령을 실행한다. 이 속도는 주로 전기적인 특성에 크게 관계되기때문에 대단히 빠르다. 그러나 디스크는 기계적이다. 그의 속도는 디스크를 회전시키고 디스크자두를 이동하는 시간에 크게 관계된다. 많은 디스크들은 3,600RPM으로 회전한다(더 빠른 디스크는 7,200RPM으로 회전한다). 따라서 1분동안에 그것은 3,600번 회전한다. 때문에 한번 회전하는데 1/60s 또는 16.7ms가 걸린다. 디스크에서 어떤 파일을 찾기 위하여 평균적으로 디스크의 절반을 회전시켜야 하는데 만일 다른 인자들을 무시하면 8.3ms초에 접근하게 된다(이것은 아주 정확한 평가인데 9~11ms에 접근하는것이 더 일반적이다). 따라서 매 초당 대략 120회의 디스크접근을 수행할수 있다. 이것은 처리기의 속도와 비교하지 않는한 상당히 좋은것이다. 120회의 디스크접근을 2천500만개의 명령에 대응시켜 보자. 그러면 한번의 디스크접근은 대략 20만개의 명령에 대응된다. 물론 여기서 모든것은 대략적인 계산이지만 상대적인 속도는 상당히 정확하다. 즉 디스크접근들은 믿기 어려울 정도로 비

실용적이다. 더우기 처리기속도들은 디스크속도(그것은 매우 빨리 증가하고 있는 디스크 크기이다.)보다 훨씬 빠른 비률로 증가하고 있다. 그러므로 한번의 디스크접근을 줄이기 위하여 노력을 아끼지 않고 있다. 거의 모든 경우에 실행시간을 좌지우지하는것은 디스크접근회수이다. 따라서 디스크접근회수를 절반으로 줄이면 그 실행시간도 절반으로 줄어 든다.

여기에 디스크에 대하여 전형적인 탐색나무를 실현하는 방법이 있다. 즉 플로리다(Florida)주 시민들에 대한 운전기록에 접근한다고 하자. 항목수는 10,000,000개이고 매 열쇠는 32byte(이름을 표현하기 위해서)이며 하나의 기록은 256byte라고 하자. 이것은 주기억기에 넣을수 없다. 또한 이 체계를 20명의 사용자가 리용한다고 하자(즉 매 사용자는 체계자원의 1/20을 가진다.). 따라서 1s에 매 사용자는 백만개의 명령을 실행하거나 6번의 디스크접근을 수행할수 있다.

불균형2진탐색나무에서 이것은 큰 일이다. 최악의 경우에 그 나무는 선형깊이를 가지며 따라서 100만번의 디스크접근을 요구하게 된다. 보통 성공적인 탐색은 1.38log*N*번의 디스크접근을 요구하며 log10000000≈24이므로 일반탐색은 32번의 디스크접근 또는 5s의 시간을 요구한다. 전형적인 우연적으로 구축된 나무에서 일부 매듭들은 3배 더 깊어 질수 있는데 이것은 대략 100번정도의 디스크접근 또는 16s의 시간을 요구한다. AVL나무는 효률이 얼마간 더 좋다. 1.44log*N*인 최악의 경우는 잘 발생하지 않으며 일반적인 경우는 log*N*에 아주 가깝다. 따라서 AVL나무는 평균 25번정도의 디스크 접근 또는 4s의 시간을 요구한다.

사용자는 디스크접근수를 3이나 4와 같은 아주 작은 상수로 줄일것을 요구한다. 이 요구를 충족시키기 위해서는 코드가 복잡해 지는것도 마다하지 않는다(기계어명령들은 기본상 자유롭게 쓸수 있다). 전형적인 AVL나무는 최적높이에 가깝기때문에 대체로 2진탐색나무는 동작하지 않는다는것이 명백하다. 2진탐색나무를 리용하면 그 실행시간은 log*N*이하로 될수 없다. 그 해결책은 직관적으로 보면 간단하다. 즉 가지가 많으면 그만큼 나무의 높이는 작아 진다. 따라서 31개의 매듭을 가진 완전2진나무는 5개 준위를 가지지만 그림 4-40에서 보여 준것처럼 31개의 매듭들을 가진 5갈래나무는 3개 준위만을 가진다. *M*갈래탐색나무는 *M*개의 가지를 가질수 있다. 가지가 증가하는것만큼 깊이는 감소된다. 완전2진나무는 대략 $\log_2 N$의 높이를 가지지만 완전5갈래나무는 약 $\log_M N$의 높이를 가진다.

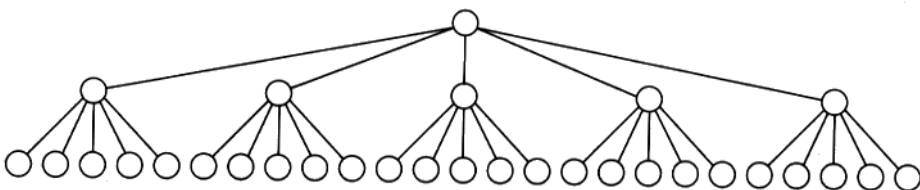



그림 4-40. 깊이가 3인 31개 매듭으로 이루어 진 5갈래나무

2진탐색나무에서와 류사한 방법으로 M갈래탐색나무를 만들수 있다. 2진탐색나무에서는 두개의 가지들가운데서 어느 가지를 선택하겠는가 하는데 한개의 열쇠가 필요하다. 그러나 M갈래탐색나무에서는 가지선택을 위하여 M-1개의 열쇠가 필요하다. 이 방안이 최악의 경우에 효과를 나타내자면 M갈래탐색나무가 어떤 방법으로 균형되여야 한다. 그렇게 하지 않으면 2진탐색나무와 같이 련결목록으로 퇴화되고 만다. 실천적으로는 좀 더 제한적인 균형조건이 요구된다. 말하자면 2진탐색나무가 log$N$번의 접근으로 고착되기때문에 M갈래탐색나무가 2진탐색나무로 퇴화되는것을 바라지 않는다.

이것을 실현하기 위한 한가지 방법이 B-나무를 리용하는것이다. 여기서는 기초적인 B-나무[15]를 서술한다. B-나무에는 많은 변종들과 개선들이 알려 져 있는데 몇가지 경우가 있으므로 그 실현은 좀 복잡하다. 그러나 원리적으로 B-나무는 적은 수의 디스크접근만을 발생한다.

M갈래의 B-나무는 다음의 성질을 가지는 M갈래나무이다.[16]

① 자료항목들은 잎매듭에 보관된다.

② 비잎매듭들은 탐색을 안내하기 위하여 M-1개까지의 열쇠를 가진다. 즉 열쇠 i는 부분나무 i+1에서 가장 작은 열쇠를 나타낸다.

③ 뿌리는 잎매듭인 경우를 제외하고 2～M개의 자식들을 가진다.

④ 모든 비잎매듭(뿌리를 제외하고)들은 $\lceil M/2 \rceil$～M개의 자식들을 가진다.

⑤ 모든 잎매듭들은 같은 깊이에 있으며 임의의 L에 대하여 $\lceil L/2 \rceil$～L개의 자식들을 가진다(L을 결정하는것은 간단히 서술한다).

5갈래 B-나무의 실례를 그림 4-41에서 보여 준다.

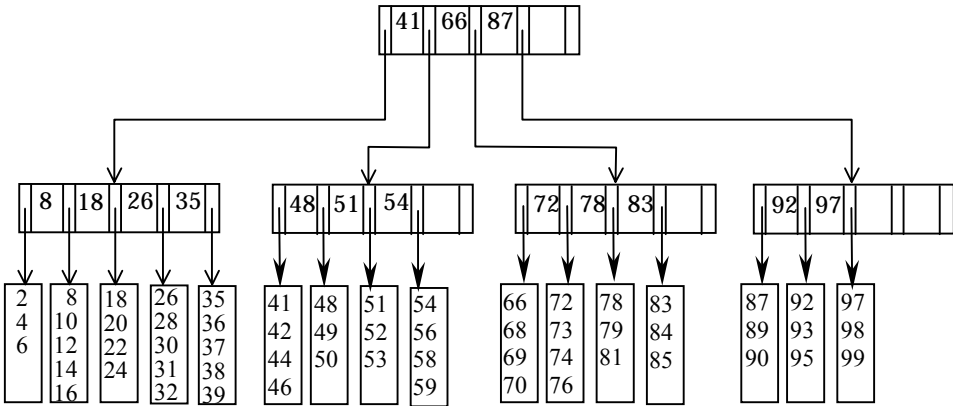



그림 4-41. 5갈래의 B-나무

---

15 여기에서 서술된것은 일반적으로 B$^+$나무로 알려 져 있다.

16 규칙 3과 5는 처음의 L번의 삽입들에 대해서는 완화되여야 한다.

모든 비잎매듭들이 3∼5개의 자식들을 가지는데(따라서 2∼4개의 열쇠를 가진다.) 뿌리는 2개의 자식만을 가질수 있다. 여기서 L=5이다. 이 실례에서는 L과 M이 같지만 꼭 그럴 필요는 없다. L이 5이므로 매개 잎매듭은 3∼5개의 자료항목들을 가진다. 매듭들이 절반정도 차 있도록 하면 B-나무가 단순한 2진나무로 퇴화되지 않는다. B-나무에 대하여 이 구조를 변경시키는 여러가지 정의들이 있다고 해도 이 정의는 일반적인 형태들중의 하나로 된다.

매개 매듭은 디스크블로크를 나타내며 따라서 보관되는 항목들의 크기에 기초하여 M과 L을 선택한다. 실례로 한개 블로크가 8,192byte라고 하자. 플로리다주 실례에서 매개 열쇠는 32byte를 리용한다. M갈래B-나무에서는 32Mbyte~32byte전체에 대하여 M−1개의 열쇠와 M개의 가지들을 가진다. 매개 가지는 다른 디스크블로크의 번호이므로 하나의 가지가 4byte이라는것을 가정할수 있다. 따라서 가지들은 4Mbyte를 리용한다. 비잎매듭을 위한 전체 기억기요구는 36Mbyte~32byte이다. 8,192를 넘지 않는 가장 큰 M값은 228이다. 따라서 M=228을 선택한다. 매개 자료기록이 256byte이므로 한개의 블로크에 32개의 기록들을 넣을수 있다. 따라서 L=32를 선택한다. 여기서는 매개 잎이 16~32개의 자료기록들을 가지며 뿌리를 제외한 모든 내부매듭은 각각 적어도 114개의 가지를 가진다고 본다. 10,000,000개의 기록들이 있으면 많아서 625,000개의 잎매듭들이 있게 된다. 그 결과 최악의 경우에 잎매듭들은 4준위에 있게 된다. 구체적으로 말하여 최악의 경우의 접근수는 근사적으로 $\log_{M/2}N \pm 1$로 주어 진다(실례로 뿌리와 첫번째 준위의 매듭들은 주기억기에 보관되여 결국 디스크접근들은 3이상의 준위에 대해서만 필요된다.).

이제 남은 문제는 B-나무에 항목들을 삽입하거나 삭제하는 방법이다. 이 방법들을 간단히 고찰하자.

먼저 삽입에 대하여 보자. 그림 4-41의 B-나무에 57을 삽입한다고 하자. 나무를 따라 내려 가면서 그 요소가 이미 나무에 있지 않는가를 조사한다. 57은 잎매듭에 다섯번째 자식으로서 추가할수 있다. 이때 그 요소를 삽입하기 위하여 그 잎매듭의 모든 자료들을 재배치하여야 한다. 그러나 이 재배치시간은 디스크접근시간에 비하면 보잘것 없는 량이다(이 경우에 디스크쓰기도 포함한다.).

물론 그 잎매듭이 아직 다 차지 않았기때문에 삽입은 상대적으로 쉽다. 이제 55를 삽입한다고 하자. 그림 4-42는 이때의 과정을 보여 주는데 55가 삽입되여야 할 잎매듭은 이미 가득 차 있다. 그 해결방법은 간단하다. 현재 L+1개의 항목들을 가지기때문에 그 잎매듭을 두개의 잎매듭으로 가르는데 이 잎매듭들은 둘다 필요한 자료기록의 최소개수를 가져야 한다. 따라서 두개의 잎매듭들이 각각 3개의 항목들을 가지도록 구성한다. 이 잎매듭들을 쓰는데 2번의 디스크접근이 요구되며 그 부모를 수정하는데 3번의 디스크접근이 요구된다. 부모매듭에서는 열쇠들과 가지들이 다 수정되여야 하는데 그 조작은 쉽게 계산되는 조종방법으로 진행한다. 그 결과로 주어 진 B-나무를 그림 4-43에 보여 준다.

매듭가르기가 적어도 2번의 추가적인 디스크쓰기를 요구하므로 시간소비가 있기는

하지만 그것은 상대적으로 드물게 발생한다. 실례로 L이 32이면 어떤 매듭을 가를 때 각각 16개와 17개의 항목들을 가지는 2개의 잎매듭이 만들어 진다. 17개의 항목을 가지는 잎매듭에 대하여 그 매듭을 더 이상 가르지 않고도 15개의 항목을 더 삽입할수 있다. 다시말하여 매듭을 한번 가르면 약 $L/2$회정도는 그 잎매듭을 가르지 않고 새로운 항목을 삽입할수 있다.

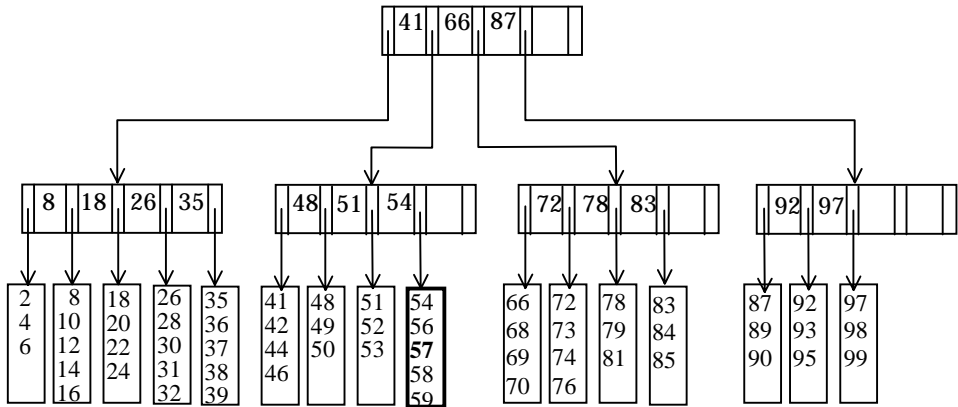



그림 4-42. 그림 4-39의 나무에 57을 삽입한후의 B-나무

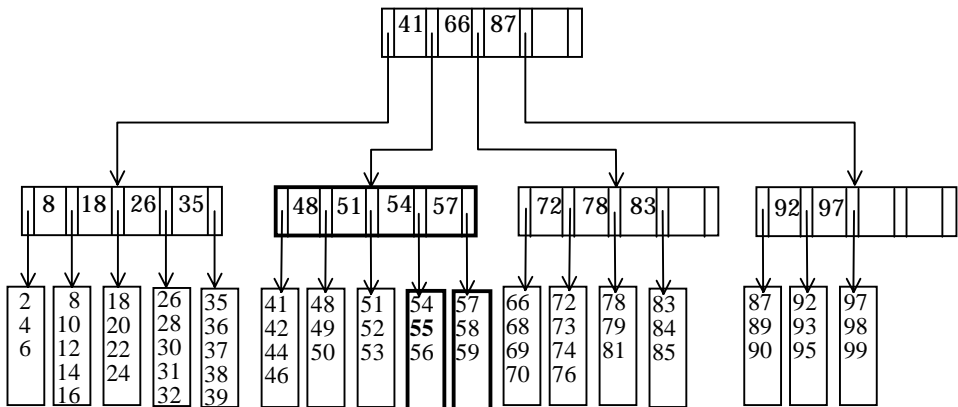



그림 4-43. 그림 4-42의 B-나무에 55를 삽입하면 매듭의 가르기가 진행된다.

앞의 실례와 같은 매듭가르기는 부모매듭이 최대한계의 자식을 가지고 있지 않기때문에 일어 난다. 그러나 만일 부모매듭이 이미 최대한계의 자식을 가지고 있다면 어떻게 되겠는가? 실례로 그림 4-43의 B-나무에 40을 삽입한다고 하자. 이때 35～39 그리고 현재의 40을 열쇠로 가지는 잎매듭을 2개의 잎매듭으로 갈라야 한다. 그러면 이 처리는 그 부모매듭에 6개의 자식을 할당하게 되는데 부모매듭은 다만 5개의 자식만을 가질수 있다. 해결책은 그 부모매듭을 가르는것이다. 그 결과를 그림 4-44에서 보여 준다. 부모매듭이

갈라 질 때 그의 열쇠값들과 그 부모매듭의 부모매듭도 갱신해야 하며 따라서 추가적인 2번의 디스크쓰기를 해야 한다(이러한 삽입은 5번의 디스크쓰기를 해야 한다). 그러나 다시 한번 고찰해 보면 코드는 여러가지 인자들때문에 확실히 복잡해 지지만 그 열쇠들은 아주 세련된 방법으로 변경된다.

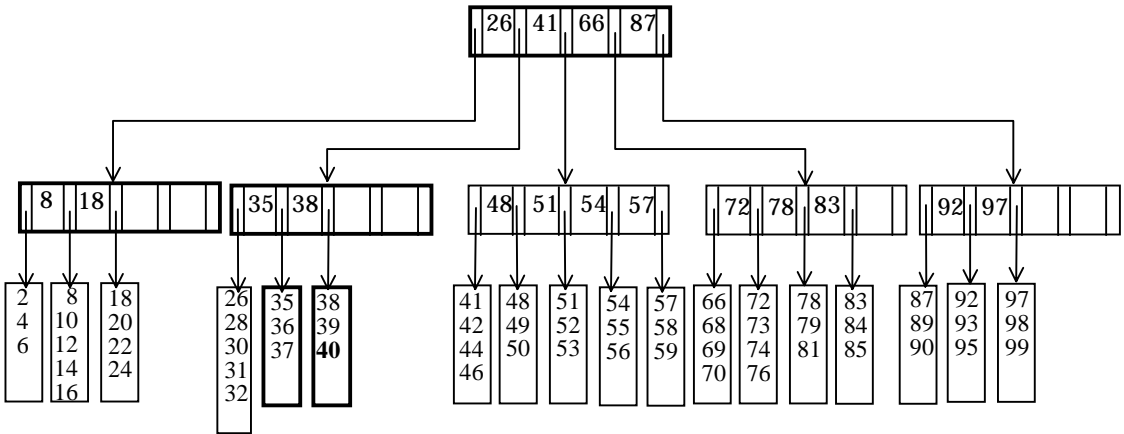



그림 4-44. 그림 4-43의 B-나무에 40을 삽입하면 잎매듭을 2개의 잎매듭으로 가르고 또 그의 부모매듭을 가르게 된다.

비잎매듭들이 우에서처럼 갈라 질 때 그의 부모매듭은 하나의 자식을 얻는다. 부모매듭이 이미 최대한계의 자식들을 가지고 있다면 어떻게 하겠는가? 그때는 가를 필요가 없는 부모매듭을 찾든가 아니면 뿌리매듭에 도달할 때까지 나무의 웃층으로 매듭을 계속 갈라 나간다. 만일 뿌리매듭을 가르면 2개의 뿌리매듭이 생긴다. 명백하게 이것은 용납될수 없는것이지만 갈라 진 두개의 뿌리매듭을 자식으로 하여 새로운 뿌리매듭을 만들수 있다. 이것이 바로 뿌리매듭이 특별히 2개의 자식을 최소로 가지게 되는 리유이다. 그것은 또한 B-나무의 높이를 증가시키는 유일한 방법이다. 두말할것 없이 뿌리매듭까지 가면서 모든 매듭을 가르는것은 극히 보기 드문 현상이다. 그것은 4개의 준위를 가진 나무는 전체 삽입서렬에 대하여 그 뿌리매듭이 3번 갈라 진다는것을 의미하기때문이다(삭제가 진행되지 않는다고 하면). 실제상 어떤 비잎매듭의 가르기도 아주 드물다.

자식들의 자리넘침을 조종하는 다른 방법들도 있다. 한가지 방법은 린접한 잎매듭에 자리가 있으면 넘쳐 난 자식을 옮겨 넣는것이다. 실례로 그림 4-44의 B-나무에 29를 삽입하기 위하여 32를 다음 잎매듭으로 옮겨 삽입할 자리를 만든다. 이 방법은 열쇠들을 이동하여야 하기때문에 부모매듭에 대한 수정을 요구한다. 그러나 이 방법은 매듭들을 더 충만시키게 하며 장기적인 실행에서 기억공간을 유지하게 하는 경향이 있다.

삭제는 제거되여야 할 항목을 찾고 그다음 그것을 제거하는 방법으로 수행할수 있다. 이때 문제로 되는것은 삭제할 항목이 존재하는 잎매듭에 최소개수의 자료항목들이 있었다면 그 수가 최소값아래로 작아 지게 된다는것이다. 만일 린접매듭에 최소개수보다 더

많은 자료항목이 있으면 그 린접매듭의 항목을 리용하여 이 상태를 수정할수 있다. 그러자면 충만된 잎매듭을 만들기 위하여 그 린접매듭과 결합해야 한다. 그러나 이것은 그 부모매듭이 하나의 자식을 잃어 버린다는것을 의미한다. 만일 부모매듭에서도 항목들의 수가 최소개수아래로 떨어 지게 되면 역시 우와 같은 방법으로 수정한다. 이러한 처리는 뿌리매듭에 접근하는 모든 통로들을 거치면서 진행될수 있다. 그런데 뿌리매듭은 하나의 자식을 가질수 없다(이것은 허용될수 없는것이다). 만일 뿌리매듭이 처리의 결과로써 하나의 자식만 남게 되면 그 뿌리매듭을 제거하고 그의 자식을 나무의 새로운 뿌리매듭으로 만든다. 이것은 B-나무에서 높이를 줄이는 유일한 방법이다. 실례로 그림 4-44의 B-나무로부터 99를 삭제한다고 하자. 잎매듭이 2개의 항목만을 가지고 그의 린접매듭의 자식수가 이미 최소값 3에 도달했으므로 그 항목들을 새로운 5개의 항목을 가진 잎매듭에 결합한다. 결과 그의 부모매듭은 2개의 자식만을 가지게 된다. 그러나 린접매듭이 4개의 자식들을 가지기때문에 린접매듭으로부터 채용할수 있다. 결과적으로 둘다 3개의 자식들을 가진다. 그 결과를 그림 4-45에 보여 준다.

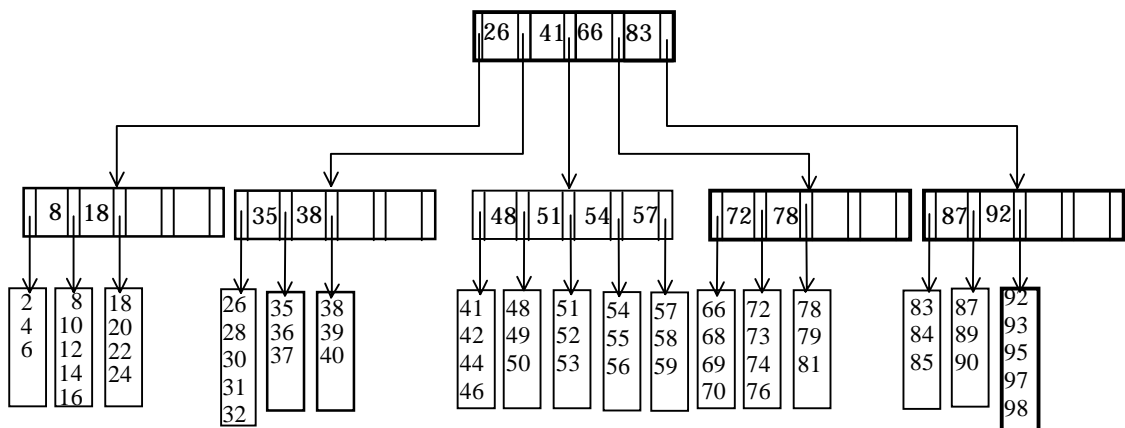



그림 4-45. 그림 4-44의 B-나무로부터 99를 삭제한 후의 B-나무

요약

이 장에서는 조작체계들과 번역기설계, 탐색에서 나무의 리용을 보았다.

수식나무들은 **구문해석나무**(*parse tree*)로 알려 진 일반적인 자료구조의 작은 실례인데 이것은 번역기설계에서 중심적인 자료구조이다. 구문해석나무들은 2진나무가 아니지만 상대적으로 수식나무들의 간단한 확장들이다(그것들을 구축하는 알고리듬이 그렇게 간단하지 않아도).

탐색나무들은 알고리듬설계에서 아주 중요하다. 그것들은 거의 다 쓸모 있는 연산들을 제공하며 그것의 로그적인 평균값은 대단히 작다. 탐색나무들의 비재귀적인 실현도

어느 정도 빠르지만 재귀적인 실현은 더 원활하고 품위가 있으며 리해와 오유수정이 쉽다. 탐색나무와 관련한 문제는 그의 성능이 우연적인 입력에 크게 관계된다는것이다. 만일 그렇지 않다면 탐색나무들은 비용이 큰 련결목록으로 된다는 점에서 실행시간이 대폭 증가한다.

이 장에서는 이 문제를 처리하는 여러가지 방법들을 보았다. AVL나무들은 모든 매듭들의 왼쪽과 오른쪽 부분나무들의 깊이가 많아서 1만큼 차이 난다는것을 강조하여 처리한다. 이것은 그 나무가 너무 깊어 지지 않게 한다. 삽입할 때 나무의 구조를 변경시키지 않는 연산들은 표준적인 2진탐색나무코드를 모두 리용할수 있다. 나무의 구조를 변경시키는 연산들은 그 나무를 다시 보관하여야 한다. 이것은 좀 복잡한데 특히 삭제경우에 더 복잡하다. 이 장에서는 $O(\log N)$시간에 어떠한 매듭을 삽입한 다음에 나무를 다시 보관하는 방법을 고찰하였다.

또한 펼친나무를 고찰하였다. 펼친나무에서 매듭들은 임의의 깊이를 가질수 있지만 매개 접근후에 그 나무는 좀 모호한 방법으로 조정된다. 종국적인 효과는 어떤 M개 연산들의 서렬이 $O(M\log N)$시간을 가지는것인데 이것은 균형나무에서와 같은 값이다.

B-나무는 균형M갈래(2갈래 또는 2진에 대조되는것으로서)나무들인데 이것은 디스크조작에 적당한 자료구조이다. 특수한 경우는 2-3나무(M=3)인데 이것은 균형탐색나무를 실현하는 또 하나의 방법이다.

실천적으로 모든 균형나무체계들에서 간단한 2진탐색나무에서보다 탐색연산의 실행시간이 좀 더 빠르고 삽입과 삭제연산의 실행시간이 나쁘지만(상수결수만큼) 이것은 자주 발생하는 최악의 경우 입력에 대응하기 위한 보호의 관점에서는 용납될수 있다. 제12장에서는 일부 추가적인 탐색나무자료구조들을 설명하고 구체적인 실현방법들을 보여 준다.

마지막주의점: 탐색나무에 요소들을 삽입하고 중뿌리순회를 수행하여 요소들을 정렬된 순서로 얻을수 있다. 이것은 정렬할 때 $O(N\log N)$알고리듬을 주는데 이 값은 그 어떤 복잡한 탐색나무가 리용되여도 최악의 경우의 시간한계이다. 제7장에서 이에 대한 더 좋은 방법을 보게 되지만 시간한계는 더 작아 지지 않는다.

련습문제

4-1부터 **4-3**까지의 문제는 그림 4-46의 나무를 참고하시오.

4-1. 그림 4-46의 나무에서

ㄱ. 뿌리매듭은 어느 매듭인가?

ㄴ. 잎매듭들은 어느 매듭들인가?

4-2. 그림 4-46의 나무의 매개 매듭에 대하여

ㄱ. 부모매듭의 이름을 말하시오.

ㄴ. 자식매듭들을 쓰시오.

ㄷ. 형제매듭들을 쓰시오.

ㄹ. 깊이를 계산하시오.

ㅁ. 높이를 계산하시오.

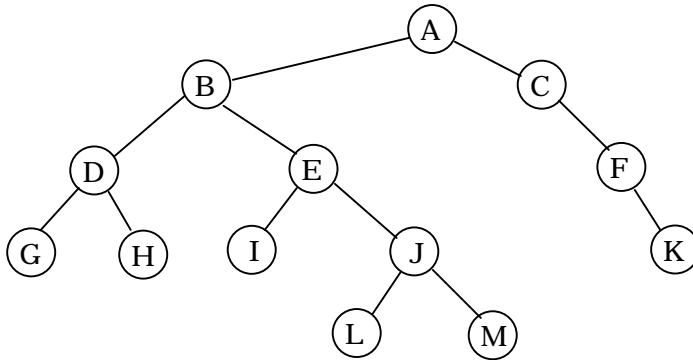



그림 4-46. 련습문제 4-1부터 4-3에 대한 나무

4-3. 그림 4-46에서 나무의 깊이는 얼마인가?

4-4. N개의 매듭들을 가지는 2진나무에서 자식들을 표현하는 N+1개의 NULL련결이 존재한다는것을 설명하시오.

4-5. 높이 h를 가지는 2진나무에서 매듭들의 최대수가 $2^{h+1}-1$이라는것을 설명하시오.

4-6. 충만매듭은 2개의 자식을 가지는 매듭이다. 충만매듭들의 수에 1을 더한 값은 비지 않은 2진나무에서 잎매듭들의 수와 같다는것을 증명하시오.

4-7. 2진나무가 깊이 d_1, $d_2, \cdots$, d_m에서 각각 l_1, $l_2, \cdots$, l_m개의 잎매듭들을 가진다고 하자. $\sum_{i=1}^{M} 2^{-d_i} \le 1$을 증명하고 같아 지는 경우를 결정하시오.

4-8. 그림 4-47의 나무의 대응하는 선뿌리순회, 중뿌리순회, 후뿌리순회식들을 쓰시오.

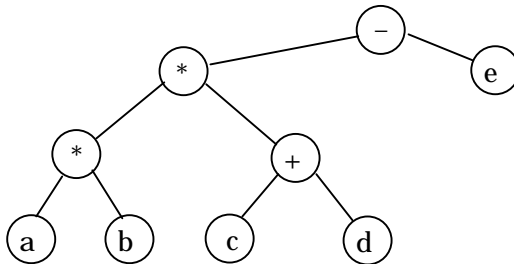



그림 4-47. 련습문제 4-8에 대한 나무

4-9. ㄱ. 초기의 빈 2진탐색나무에 3, 1, 4, 6, 9, 2, 5, 7을 삽입한 결과를 표시

하시오.

ㄴ. 뿌리매듭을 삭제한 결과를 표시하시오.

4-10. 2진탐색나무에서 충만매듭들의 평균수를 $f(N)$이라고 하자.

ㄱ. $f(0)$과 $f(1)$의 값을 결정하시오.

ㄴ. N>1일 때

$$f(N)=\frac{N-2}{N}+\frac{1}{N}\sum_{i=0}^{N-1}(f(i)+f(N-i-1))$$ 을 증명하시오.

ㄷ. $f(N)=(N-2)/3$이 ㄱ의 초기조건을 가지고 ㄴ의 방정식에 대한 풀이라는것을 증명하시오(유도법에 의해서).

ㄹ. 2진탐색나무에서 잎들의 평균수를 결정하시오. 련습문제 4-6의 결과를 리용하시오.

4-11. 2진탐색나무들은 유표련결목록실현과 류사한 방법을 리용하여 유표를 가지고 실현될수 있다. 유표실현을 리용하여 기본2진탐색나무루틴들을 서술하시오.

4-12. 련결목록클라스를 가지고 처리한것처럼 find(그리고 findMin과 findMax)의 결과를 되돌리는 반복기를 정의하여 2진탐색나무를 수정하시오. 반복기는 current를 보관한다. 대응하는 값은 retrieve에 의해 접근될수 있다. current가 NULL이 아니면 참으로 되는 isValid를 제공하는 반복기를 실현하시오.

4-13. 현재매듭에 대한 접근경로를 보관하는 탄창을 추가하여 련습문제 4-12의 반복기를 확장하시오. 이 방법으로 first와 advance를 실현할수 있다.

4-14. 우연적인 insert/remove쌍들에 의해서 발생될수 있는 문제를 검증하기 위한 실험을 하려고 한다. 여기에 완전히 우연적이지는 않지만 충분히 가까운 방법이 있다. 1부터 $M=\alpha N$까지의 범위에서 우연적으로 선택된 N개의 요소들을 삽입하여 N개의 요소들을 가진 나무를 구축한다. 그다음 N^2개의 삽입후 삭제쌍들을 수행한다. a와 b를 포함하여 그사이에서 고정적인 임의의 옹근수를 되돌리는 루틴 randomInteger(a, b)가 있다고 하자.

ㄱ. 나무에 아직 없는(그래서 우연적인 삽입이 수행될수 있는) 1과 M사이의 우연적인 옹근수를 발생하는 방법을 설명하시오. N과 α로 환산하면 이 연산의 실행시간은 얼마인가?

ㄴ. 나무에 이미 존재하는(그래서 우연적인 삭제가 수행될수 있는) 1과 M사이의 우연적인 옹근수를 발생하는 방법을 설명하시오. 이 연산의 실행시간은 얼마인가?

ㄷ. α를 어떻게 선택하면 좋은가? 왜 그런가?

4-15. 2개의 자식들을 가지는 매듭을 삭제하기 위한 다음의 방법들을 경험적으로

평가하는 프로그람을 작성하시오.

ㄱ. T_L에서 가장 큰 매듭 X로 치환하고 재귀적으로 X를 삭제한다.

ㄴ. T_L에서 가장 큰 매듭과 T_R에서 가장 작은 매듭으로 번갈아 치환하고 적당한 매듭을 재귀적으로 삭제한다.

ㄷ. T_L에서 가장 큰 매듭 또는 T_R에서 가장 작은 매듭을 우연적으로 선택하여 치환하고 적당한 매듭을 재귀적으로 삭제한다. 어느 방법이 가장 좋은 평형을 보장하는가? 전체 서렬을 처리하는데 어느것이 가장 짧은 CPU시간을 가지는가?

4-16. 지연삭제를 실현하기 위하여 2진탐색나무클라스를 고쳐 쓰시오. 이것이 모든 루틴들에 영향을 준다는데 주의하시오. 특별히 주목할것은 findMin과 findMax인데 이것은 재귀적으로 처리되여야 한다.

****4-17.** 우연적인 2진탐색나무의 깊이(가장 깊은 매듭의 깊이)가 평균 $O(\log N)$임을 증명하시오.

4-18. *ㄱ. 높이가 h인 AVL나무에서 매듭들의 최소개수에 대한 정확한 표현을 주시오.

ㄴ. 높이가 15인 AVL나무에 매듭들의 최소개수는 얼마인가?

4-19. 초기의 빈 AVL나무에 2, 1, 4, 5, 9, 3, 6, 7을 삽입한 결과를 표시하시오.

***4-20.** 초기의 빈 AVL나무에 열쇠 1, 2, …, 2^k-1이 차례로 삽입되였다. 얻어 지는 나무가 완전히 평형이라는것을 증명하시오.

4-21. AVL나무에서 단일회전, 2중회전을 실현하기 위한 나머지처리를 서술하시오.

4-22. AVL나무에서 높이정보가 정확히 유지되고 균형성질이 순차적이라는것을 검증하는 선형시간알고리듬을 설계하시오.

4-23. AVL나무에 삽입하기 위한 비재귀함수를 서술하시오.

***4-24.** AVL나무에서 비지연삭제를 어떻게 실현할수 있는가?

4-25. ㄱ. N개 매듭을 가지는 AVL나무에서 매듭의 높이를 보관하는데 매 매듭당 몇비트가 요구되는가?

ㄴ. 높이가 8bit값을 넘는 가장 작은 AVL나무의 높이는 얼마인가?

4-26. 2개의 단일회전들을 진행하지 않고 2중회전을 수행하는 함수를 서술하시오.

4-27. 그림 4-48에 있는 펼친나무에서 열쇠 3, 9, 1, 5에 차례로 접근할 때의 결과를 표시하시오.

4-28. 우의 문제에 대하여 얻어 지는 펼친나무에서 열쇠 6을 가진 요소를 삭제한 결과를 표시하시오.

4-29. ㄱ. 펼친나무의 모든 매듭들이 순차적으로 접근되면 결과적인 나무는 왼쪽 자식들에 대한 사슬들로 구성된다는것을 나타내시오.

**ㄴ. 펼친나무의 모든 매듭들이 순차적으로 접근되면 초기나무에 관계없이

전체 접근시간이 $O(N)$이라는것을 증명하시오.

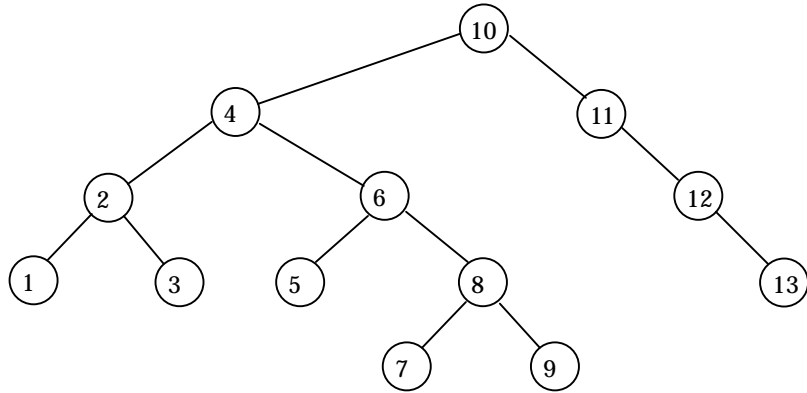



그림 4-48. 련습문제 4-27을 위한 나무

4-30. 펼친나무에 대하여 우연적인 연산들을 수행하는 프로그람을 작성하시오. 그 전과정에 수행되는 총 회전수를 계수하시오. 그 실행시간을 AVL나무와 불균형2진탐색나무에서의 실행시간과 어떻게 비교하는가?

4-31. 2진나무 T의 뿌리매듭에 대한 지적자만을 가지고 다음의것들을 계산하는 효과적인 함수들을 서술하시오.

ㄱ. T에 있는 매듭들의 수

ㄴ. T에 있는 잎매듭들의 수

ㄷ. T에 있는 충만매듭들의 수

이 루틴들의 실행시간은 얼마인가?

4-32. 2진나무가 모든 매듭에서 탐색나무순서성을 만족시키는가를 검사하는 재귀적인 선형시간알고리듬을 작성하시오.

4-33. 나무 T의 뿌리매듭에 대한 지적자를 가지고 T의 모든 잎매듭들을 제거하여 얻은 결과나무의 뿌리매듭에 대한 지적자를 되돌리는 재귀적인 함수를 서술하시오.

4-34. 1부터 N까지의 서로 다른 열쇠들을 가지고 N개의 매듭을 가지는 우연적인 2진탐색나무를 만드는 함수를 작성하시오. 그 함수의 실행시간은 얼마인가?

4-35. 높이가 h이고 가장 적은 수의 매듭을 가지는 AVL나무를 만드는 함수를 작성하시오. 그 함수의 실행시간은 얼마인가?

4-36. 1부터 2^{h+1}-1까지의 열쇠들을 가지고 높이가 h인 완전균형2진탐색나무를 만드는 함수를 작성하시오. 그 함수의 실행시간은 얼마인가?

4-37. 입력으로서 2진탐색나무 T와 두개의 열쇠단어 k_1와 k_2($k_1 \leqq k_2$)을 가지고 그 나무에서 $k_1 \leqq Key(X) \leqq k_2$인 요소 X를 모두 출력하는 함수를 서술하시오. 열쇠들

이 시종일관 순서화될수 있다는것을 제외하고 열쇠들의 형에 대한 그 어떤 정보도 고려하지 않는다. 그 프로그람은 $O(K+\log N)$평균시간에 실행되여야 한다(여기서 K는 출력되는 열쇠들의 개수). 알고리듬의 실행시간을 경계 지으시오.

4-38. 이 장에 있는 보다 큰 2진나무들은 프로그람에 의해서 자동적으로 만들어 진다. 이것은 매개 나무매듭에 (x, y)자리표를 할당하고 매개 자리표에 원주를 그리며(이것을 도형이라고 보기는 어렵다.) 매개 매듭을 그의 부모매듭에 련결하는 방법으로 수행된다. 기억기에 보관된 2진탐색나무(물론 우에서 언급한 루틴들가운데서 어느 하나에 의해 만들어 진)가 있고 매개 매듭이 자리표들을 보관하기 위한 2개의 추가마당을 가진다고 가정한다.

ㄱ. x자리표는 중뿌리순회번호를 할당하여 계산될수 있다. 나무의 매개 매듭에서 이것을 수행하는 루틴을 서술하시오.

ㄴ. y자리표는 매듭깊이의 부수값을 리용하여 계산될수 있다. 나무의 매개 매듭에서 이것을 수행하는 루틴을 서술하시오.

ㄷ. 어떤 가상단위로 환산할 때 도형의 차원은 얼마인가? 나무의 높이가 항상 너비의 약 2～3배로 되도록 하자면 단위들을 어떻게 조절할수 있는가?

ㄹ. 선의 교차가 없는 이 체계를 리용하면 임의의 매듭 X에 대하여 X의 왼쪽 부분나무에 있는 모든 요소들은 X의 왼쪽에 나타나고 X의 오른쪽 부분나무에 있는 모든 요소들은 X의 오른쪽에 나타난다는것을 증명하시오.

4-39. 어떤 나무를 다음과 같은 도형기호명령들로 변환하는 일반용나무그리기프로그람을 서술하시오.

ㄱ. *Circle*(X,Y)

ㄴ. *DrawLine*(i, j)

첫번째 명령은 (X, Y)에 원주를 그리며 두번째 명령은 i번째 원주를 j번째 원주에 련결한다(원주들은 그려진 순서로 번호를 붙인다). 이제 이것을 프로그람으로 작성하고 입력언어의 정렬을 정의하든지 아니면 다른 프로그람으로부터 호출될수 있는 함수를 서술하든지 하여야 한다. 그 루틴의 실행시간은 얼마인가?

4-40. 준위순서순회로 2진나무의 매듭들을 렬거하는 루틴을 서술하시오. 먼저 뿌리매듭을 표시하고 다음에 1준위의 매듭들을 표시하고 그다음 2준위의 매듭들을 표시하는 방법으로 처리하시오. 이러한 처리를 선형적인 시간내에 수행해야 한다. 그 시간한계를 증명하시오.

4-41. *ㄱ. B-나무에서 매듭을 삽입하는 루틴을 서술하시오.

*ㄴ. B-나무로부터 매듭을 삭제하는 루틴을 서술하시오. 항목이 삭제될 때 내부매듭들에서 정보를 갱신하는것이 필요한가?

*ㄷ. 만일 이미 M개의 항목을 가지는 매듭에 추가하려고 하면 매듭을 가르기전에 M보다 작은 수의 자식을 가지는 형제매듭의 탐색이 수행되도록 삽입루틴을 수정하시오.

4-42. M갈래 B^*-나무는 매개 매듭이 $2M/3$～M개사이의 자식들을 가지는 B-나무이다. B^*-나무에 매듭을 삽입하는 산법을 설명하시오.

4-43. 그림 4-49에 있는 나무를 자식-형제련결법으로 표현하시오.

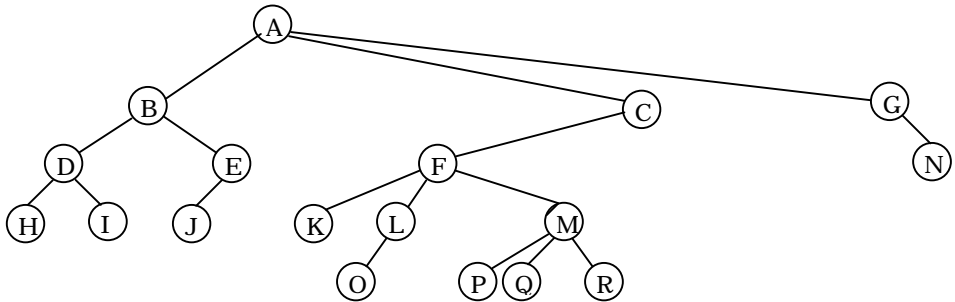



그림 4-49. 련습문제 4-43을 위한 나무

4-44. 자식-형제련결을 가지고 보관된 나무를 순회하는 수속을 서술하시오.

4-45. 만일 두개의 2진나무들이 둘다 비였거나 둘다 비지 않았으며 왼쪽과 오른쪽 부분나무들이 류사하다면 그 두 2진나무는 류사하다. 두개의 2진나무가 류사한가를 결정하는 함수를 서술하시오. 그 함수의 실행시간은 얼마인가?

4-46. 두개의 나무 T_1과 T_2이 있어서 T_1의 매듭들(또는 그 일부)의 왼쪽과 오른쪽 자식들을 서로 바꾸어 T_1가 T_2로 전환될수 있을 때 이 두 나무는 동형(isomorphic)이다. 실례로 그림 4-50에 있는 두개의 나무들은 *A*, *B*, *G*의 자식들을 서로 바꾸면 같아 지기때문에 동형이다.

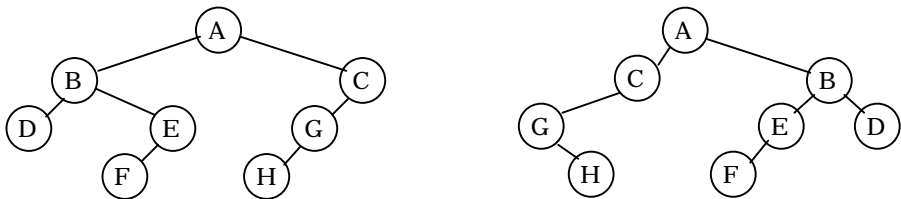



그림 4-50. 두개의 동형나무

ㄱ. 두 나무들이 동형인가를 결정하는 다차원시간을 가지는 알고리듬을 서술하시오.

*ㄴ. 그 프로그람의 실행시간은 얼마인가?(거기에는 선형풀이가 존재한다.)

4-47. *ㄱ. AVL단일회전들을 통하여 임의의 2진탐색나무 T_1가 같은 항목들을 가진 다른 탐색나무 T_2로 전환될수 있다는것을 설명하시오.

*ㄴ. 평균 $O(N\log N)$의 회전들을 리용하여 이 전환을 수행하는 알고리듬을 주시오.

**ㄷ. 이 전환이 최악의 경우 $O(N)$번의 회전으로 수행될수 있다는것을 증명하시오.

4-48. 연산 findKth를 추가한다고 하자. 연산 findKth(k)는 나무에서 k번째로 작은 항목을 되돌린다. 모든 항목들은 서로 같지 않다. 다른 연산의 시간한계들에 손실을 주지 않고 $O(\log N)$평균시간에 이 연산을 수행하도록 2진탐색나무를 수정하는 방법을 설명하시오.

4-49. N개의 매듭을 가지는 2진탐색나무는 N+1개의 NULL지적자를 가지기때문에 2진탐색나무에서 지적자정보용으로 할당된 공간의 절반이 쓸모 없다. 만일 어떤 매듭의 왼쪽 자식이 NULL이면 그 매듭의 왼쪽자식을 중뿌리순서선배에 련결하고 만일 매듭의 오른쪽 자식이 NULL이면 그 매듭의 오른쪽자식을 중뿌리순서후배에 련결하자. 이때 이것을 실끈달린나무(*threaded tree*)라고 하며 여분의 련결을 실끈(thread)이라고 한다.

ㄱ.실끈과 실지 자식에 대한 지적자를 어떻게 구별할수 있는가?

ㄴ.실끈달린나무에서 우에서 서술된 방식으로 삽입과 삭제를 수행하는 루틴들을 서술하시오.

ㄷ.실끈달린나무의 유리점은 무엇인가?

4-50. C++원천코드파일을 읽어 들이고 모든 식별자(설명문이나 문자형상수에 속하지 않는 예약어가 아닌 변수이름들)들의 목록을 자모순으로 출력하는 프로그람을 작성하시오. 매 식별자는 그것이 있는 행번호들의 목록과 함께 출력되여야 한다.

4-51. 어떤 책에 대하여 색인을 만드시오. 입력파일은 색인항목들의 모임으로 구성된다. 매개 행은 문자렬 IX:, 그 뒤에 중괄호({})로 둘러 쌓인 색인항목이름, 중괄호로 둘러 싸인 페지번호로 이루어 진다. 색인항목이름안의 매개 !는 보조준위를 나타낸다. |(는 범위의 시작을 나타내고 |)는 범위의 끝을 나타낸다. 때때로 이 범위는 같은 페지에 있을수 있다. 이 경우에 하나의 페지번호만이 출력된다. 그렇게 하지 않으면 범위들이 축소 또는 확장되지 않는다. 실례로 그림 4-51은 간단한 입력을 보여 주며 그림 4-52는 그에 대응하는 출력을 보여 준다.

```
IX:   {Series|(}               {2}
IX:   {Series!geometric|(}     {4}
IX:   {Euler's constant}       {4}
IX:   {Series!geometric|)}     {4}
IX:   {Series!arithmetic|(}    {4}
IX:   {Series!arithmetic|)}    {5}
IX:   {Series!harmonic|(}      {5}
IX:   {Euler's constant}       {5}
IX:   {Series!harmonic|)}      {5}
IX:   {Series|)}               {5}
```

그림 4-51. 련습문제 4-51을 위한 간단한 입력

```
Euler's constant: 4, 5
Series: 2-5
     arithmetic: 4-5
     geometric: 4
     harmonic: 5
```

그림 4-52. 련습문제 4-51을 위한 간단한 출력

참고문헌

2진탐색나무에 대한 그이상의 내용들, 특히 나무들의 수학적성질들은 Knuth가 쓴 책들([22],[23])에서 찾아 볼수 있다.

여러 론문들은 2진탐색나무에서 편견적인 삭제알고리듬들에 의해서 초래되는 평형의 파괴에 대하여 취급한다. Hibbard의 론문([19])은 최초의 삭제알고리듬을 제안하고 한번의 삭제가 나무의 우연성을 방지한다는것을 립증하였다. 완전한 해석은 3개 매듭([20])과 4개 매듭([5])을 가진 나무들에 대해서만 수행되였다. Eppinger의 론문([14])은 이미 비우연성에 대한 경험적인 증거를 제공하였으며 Culberson과 Munro의 론문들([10],[11])은 일부 해석적인 증거를 제공하였다(그러나 삽입과 삭제들이 혼합된 일반적인 경우에 대한 완전한 증명은 아니다).

AVL나무는 Adelson-Velskii와 Landis([1])에 의하여 제안되였다. AVL나무에 대한 모의결과들과 임의의 값 *k*에 대하여 높이차가 최대로 *k*인 변종들은 [21]에 있다. AVL나무에 대한 삭제알고리듬은 [23]에서 찾아 볼수 있다. AVL나무에서 평균탐색비용에 대한 분석은 아직 불충분한데 그 일부 결과들은 [24]에 있다.

[3]과 [8]은 제4장 제5절 1에 있는 형태와 같은 자체조정식나무를 고찰하였다. 펼친나무들은 [28]에 서술되였다.

B-나무들은 [6]에 처음으로 있다. 최초의 론문에 서술된 형식에서는 자료가 잎매듭은 물론 내부매듭에도 보관될수 있다. 그 자료구조는 B^+나무로 알려 져 있다. B-나무들의 각이한 형태에 대한 개괄은 [9]에 있다. 여러가지 방안에 대한 경험적인 결과는 [17]

에 있다. 2-3나무들과 B-나무들에 대한 분석은 [4],[13],[32]에서 찾아 볼수 있다.

련습문제 4-17은 어려운것처럼 보인다. 그 해결방안은 [15]에서 찾을수 있다. 련습문제 4-29는 [32]로부터 참고할수 있다. 련습문제 4-42에서 서술된 B*-나무에 대한 정보는 [12]에서 찾아 볼수 있다. 련습문제 4-46은 [2]에서 찾아 볼수 있다. 2*N*-6번의 회전을 리용하는 련습문제 4-47에 대한 풀이는 [29]에 주어 져 있다. 실끈의 리용(련습문제 4-49)은 [27]에서 처음으로 제안되였다. 다차원자료들을 처리하는 *k*-차원나무는 [7]에서 처음으로 제안되였으며 제12장에서 론의한다.

다른 일반적인 균형탐색나무들은 흑적나무([18])와 무게균형나무([26])이다. 그외의 균형나무형태들은 [16]과 [25],[30]에서뿐아니라 제12장에서도 찾아 볼수 있다.

1. G. M. Adelson-Velskii and E. M. Landis, "An Algorithm for the Organization of Information," *Soviet. Mat. Doklady*, 3 (l962), 1259-1263.
2. A. V. Aho, J. F. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Mass., 1974.
3. B. Allen and J. I. Munro, "Self Organizing Search Trees," *Journal of the ACM*, 25 (1978), 526-535.
4. R. A. Baeza-Yates, "Expected Behaviour of B+-trees under Random Insertions," *Acta Informatica*, 26 (1989), 439-471.
5. R. A. Baeza-Yates, "A Trivial Algorithm Whose Analysis Isn't: A Continuation," *BIT*, 29 (1989), 88-113.
6. R. Bayer and E. M. McCreight, "Organization and Maintenance of Large Ordered Indices," *Acta Informatica*, 1 (1972), 173-189.
7. J. L.,. Bentley, "Multidimensional Binary Search Trees Used for Associative Searching," *Communications of the ACM*, 18 (1975), 509-517.
8. J. R. Bitner, "Heuristics that Dynamically Organize Data Structures," *SIAM Journal on Computing*, 8 (1979), 82-110.
9. D. Comer, "The Ubiquitous B-tree," *Computing Surveys*, 11 (1979), 121-137.
10. J. Culberson and J. I. Munro, "Explaining the Behavior of Binary Search Trees under Prolonged Updates: A Model and Simulations," *Computer journal*, 32 (1989), 68-75.
11. J. Culberson and J. I. Munro, "Analysis of the Standard Deletion Algorithms in Exact Fit Domain Binary Search Trees," *Algorithmica, 5* (1990), 295-311.
12. K. Culik, T. Ottman, and D. Wood, "Dense Multiway Trees," *ACM Transactions on Database Systems, 6* (1981), 486-512.
13. B. Eisenbath, N. Ziviana, G. H. Gonnet, K. Melhorn, and D. Wood, "The Theory of Fringe Analysis and its Application to 2-3 Trees and B-trees," *Information and Control*, 55 (1982), 125-174.
14. J. L. Eppinger, "An Empirical Study of Insertion and Deletion in Binary Search Trees,"

Communications of the ACM, 26 (1983), 663-669.

15. P. Flajolet and A. Odlyzko, “The Average Height of Binary Trees and Other Simple Trees,” *Journal of Computer and System Sciences*, 25 (1982), 171-213.
16. G. H. Gonnet and R. Baeza-Yates, *Handbook of Algorithms and Data Structures*, 2d ed., Addison-Wesley, Reading, Mass., 1991.
17. E. Gudes and S. Tsur, “Experiments with B-tree Reorganization,” *Proceedings of ACM SIGMOD Symposium on Management of Data* (1980), 200-206.
18. L. J. Guibas and R. Sedgewick, “A Dichromatic Framework for Balanced Trees,” *Proceedings of the Nineteenth Annual IEEE Symposium on Foundations of Computer Science* (1978), 8-21.
19. T. H. Hibbard, “Some Combinatorial Properties of Certain Trees with Applications to Searching and Sorting,” *Journal of the ACM*, 9 (1962), 13-28.
20. A. T. Jonassen and D. E. Knuth, “A Trivial Algorithm Whose Analysis Isn't,” *Journal of Computer and System Sciences*, 16 (1978), 301-322.
21. P. L. Karlton, S. H. Fuller, R. E. Scroggs, and E. B. Kaehler, “Performance of Height Balanced Trees,” *Communications of the ACM*, 19 (1976), 23-28.
22. D. E. Knuth, *The Art of Computer Programming: Vol. 1: Fundamental Algorithms,* 3d ed., Addison-Wesley, Reading, Mass., 1997.
23. D. E. Knuth, *The Art of Computer Programming: Vol. 3: Sorting and Searching*, 2d ed., Addison-Wesley, Reading, Mass., 1998.
24. K. Melhorn, “A Partial Analysis of Height-Balanced Trees under Random Insertions and Deletions,” *SIAM Journal of Computing*, 11 (1982), 748-760.
25. K. Melhorn, *Data Structures and Algorithms 1: Sorting and Searching*, Springer-Verlag, Berlin, 1984.
26. J. Nievergelt and E. M. Reingold, “Binary Search Trees of Bounded Balance,” *SIAM Journal on Computing*, 2 (1973), 33-43,
27. A. J. Perlis and C. Thornton, “Symbol Manipulation in Threaded Lists,” *Communications of the ACM*, 3 (1960), 195-204.
28. D. D. Sleator and R. E. Tarjan, “Self-adjusting Binary Search Trees,” *Journal of the ACM*, 32 (1985), 652-686.
29. D. D. Sleator, R. E. Tarjan, and W. P. Thurston, “Rotation Distance, Triangulations, and Hyperbolic Geometry,” *Journal of the AMS* (1988), 647-682.
30. H. F. Smith, *Data Structures—Form and Function*, Harcourt Brace Jovanovich, Orlando, Fla., 1987.
31. R. E. Tarjan, “Sequential Access in Splay Trees Takes Linear Time,” *Combinatorica*, 5 (1985), 367-378.
32. A. C. Yao, “On Random 2-3 Trees,” *Acta Informatica*, 9 (1978), 159-170.

제5장. 하쉬법

제4장에서는 요소들의 모임에 대하여 여러가지 연산들을 진행할수 있는 탐색나무 ADT를 고찰하였다. 이 장에서는 2진탐색나무로 할수 있는 연산들의 부분모임만을 지원하는 **하쉬표**(hash table)ADT에 대하여 고찰한다.

하쉬표의 실현을 흔히 **하쉬법**(hashing)이라고 한다. 하쉬법은 상수평균시간안에 삽입, 삭제, 탐색연산을 실현하는데 리용되는 방법이다. 요소들에서 어떤 순서적인 정보를 요구하는 나무의 연산들은 효과적인 지원을 하지 못한다. 따라서 findMin, findMax 그리고 전체 표를 선형시간에 정렬된 순서로 출력하는것과 같은 연산들은 주지 않는다.

이 장에서 중심적인 자료구조는 하쉬표이다. 여기서

- 하쉬표를 실현하는 여러가지 방법
- 이 방법들에 대한 분석적인 비교
- 하쉬법의 여러가지 응용
- 2진탐색나무와 하쉬표의 비교

에 대하여 학습한다.

제1절. 일반적인 개념

리상적인 하쉬표자료구조는 어떤 고정된 크기를 가지는 항목들의 단순한 배렬이다. 제4장에서 설명한것처럼 일반적으로 탐색은 항목의 어떤 부분(즉 자료성원)에 대하여 실현된다. 이것을 열쇠(key)라고 한다. 실례로 하나의 항목은 열쇠단어로 리용되는 문자렬과 부가적인 자료성원으로 이루어 진다(실례로 큰 공장에서 종업원구조체의 한 부분인 종업원이름). 표의 크기는 TableSize로 참조한다. 이것은 하쉬자료구조의 한 부분으로서 단순한 어떤 대역적인 변수가 아니다. 일반적으로 표는 0부터 TableSize-1까지라고 약속한다.

매개 열쇠는 0부터 TableSize-1의 범위에 있는 어떤 수로 넘겨 져서 적당한 세포에 배치된다. 그 넘기기를 **하쉬함수**(hash function)라고 하는데 리상적으로 이것은 계산이 간단하여야 하며 임의의 두개의 서로 다른 열쇠들은 서로 다른 세포를 가져야 한다. 세포들의 개수는 제한되여 있고 열쇠들은 대체로 무진장하게 제공되므로 명백히 이것은 불가능하다. 따라서 열쇠들을 세포들에 고르롭게 할당하는 하쉬함수가 요구된다. 그림 5-1은 하쉬표에 대한 리상적인 상태를 보여 준다.

그림 5-1. 리상적인 하쉬표

이 실례에서 john은 3, phil는 4,dave는 6, mary는 7에 넘겨 진다. 이것이 하쉬법의 기본개념이다. 남은 문제들은 다만 함수를 선택하고 2개의 열쇠가 같은 값으로 넘겨 질 때(이것을 충돌이라고 한다.)의 처리와 하쉬표의 크기를 결정하는것이다.

제2절. 하쉬함수

입구열쇠가 옹근수들인 때 간단히 key mod TableSize를 되돌리는것은 일반적으로 key가 어떠한 좋지 못한 속성들을 가지지 않으면 합리적인 방법으로 된다. 이 경우에 하쉬함수의 선택은 주의깊게 고려되여야 할 필요가 있다. 실례로 표크기가 10이고 그 열쇠들이 모두 0으로 끝나면 우의 하쉬함수는 좋지 못한 선택으로 된다. 그 리유는 후에 보게 될것이다. 우에서와 같은 현상을 피하기 위하여 표크기를 어떠한 씨수로 선택하는것이 좋다. 입구열쇠들이 우연적인 옹근수들이면 이 함수는 계산하기가 아주 간단할뿐아니라 열쇠단어들을 균등하게 할당한다.

보통 열쇠들은 문자렬들인데 이 경우에 하쉬함수는 주의깊게 선택할 필요가 있다.

또 한가지 방법은 문자렬에 있는 문자들의 ASCII값들을 더하는것이다. 프로그람 5-1에 있는 루틴은 이 방법을 리용한다.

```
int hash( const string & key, int tableSize )
{
    int hashVal = 0;

    for( int i = 0; i < key.length(); i++ )
        hashVal += key[ i ];

    return hashVal % tableSize;
}
```

프로그람 5-1. 간단한 하쉬함수

프로그람 5-1에 보여 준 하쉬함수는 실현하기 간단하고 결과를 고속으로 계산한다. 그러나 표의 크기가 크면 그 함수는 열쇠단어들을 잘 할당하지 못한다. 실례로 *TableSize*=10007(10007은 씨수이다.)이라고 하자. 모든 열쇠단어가 8개이하의 문자들로 되여 있다고 하자. ASCII문자가 많아서 127개의 옹근수값을 가지기때문에 하쉬함수는 0과 127*8인 1016사이의 값들만을 가질수 있다. 이것은 명백히 적당한 분배가 아니다.

또 다른 하쉬함수를 프로그람 5-2에서 보여 준다. 이 하쉬함수는 key가 적어도 세개의 문자를 가진다고 가정한다. 값 27은 영어자모에 있는 문자들의 수에 공백을 더한 값을 나타내며 729는 27^2이다. 이 함수는 첫 3개의 문자들만 검사하지만 이것들이 우연적이고 표크기가 앞에서와 같이 10007이면 아주 적당한 분배로 된다. 그러나 영어단어들은 우연적이지 않다. 26^3=17,576이 공백을 무시하고 3개의 문자들에 대한 가능한 결합들의 수라고 하여도 적당히 큰 직결사전들에 대한 검사결과는 각이한 문자결합들의 수가 실제로는 2,851뿐이라는것을 보여 준다. 지어 이 결합들이 충돌하지 않을 때에도 표의 28%만이 실제적으로 하쉬될수 있다. 따라서 이 함수는 쉽게 계산할수 있다고 하더라도 하쉬표가 어느 정도 클 때에는 적당하지 못한것으로 된다.

```
int hash( const string & key, int tableSize )
{
   return ( key[ 0 ] + 27 * key[ 1 ] + 729 * key[ 2 ] %tableSize;
}
```

프로그람 5-2. 또다른 하쉬함수(그리 좋지 않음)

프로그람 5-3은 하쉬함수에 대한 세번째 방법을 보여 준다. 이 하쉬함수는 열쇠에 있는 모든 문자들을 포함하며 잘 분배되도록 하는데서 일반적으로 기대되는 함수이다.

```
/**
 *    A hash routine for string objects.
 */
int hash( const string & key, int tableSize )
{
    int hashVal = 0;

    for( int i = 0; i < key.1ength( ); i++ )
        hashVal = 37 * hashVal + key[ i ];

    hashVal %= tableSize;
    if( hashVal < 0 )
        hashVal += tableSize;
    return hashVal:
}
```

프로그람 5-3. 좋은 하쉬함수

(이 함수는 $\sum_{i=0}^{KeySize-1} Key[KeySize-i-1]\cdot 37^i$ 을 계산하고 그 결과를 적당한 범위로 나타낸다.) 그 코드는 37을 가지고 호너(Horner)의 규칙을 리용하여 다항식함수를 계산한다. 실례로 $h_k=k_0+37k_1+37^2k_2$을 계산하는 또 다른 방법은 식 $h_k=((k_2)*37+k_1)*37+k_0$을 리용하는것이다. 호너의 규칙은 이것을 차수 n을 가진 다항식으로 확장한다.

하쉬함수는 자리넘침이 허용된다는 우점을 가진다. 이것은 부수를 받아 들이게 될것이며 따라서 함수의 끝에서 특별한 검사를 진행하게 된다.

프로그람 5-3에 보여 준 하쉬함수는 표할당에 대해서는 가장 좋지 못하지만 아주 단순하고 상당히 빠르다. 열쇠단어가 대단히 길면 하쉬함수는 너무 길어 져서 계산할수 없게 된다. 이 경우에 일반적인 방법은 열쇠단어의 문자들을 모두 리용하지 않는것이다. 열쇠단어들의 길이와 성질들은 하쉬함수의 선택에 영향을 미친다. 실례로 열쇠단어가 어떠한 복잡한 거리의 주소이면 하쉬함수는 거리의 주소로부터 두세개의 문자들이든지 아니면 도시이름과 우편번호로부터 두세개의 문자들을 포함할수 있다. 일부 프로그람작성자들은 홀수번째 위치에 있는 문자들만을 리용하여 하쉬함수를 실현하는데 이러한 방법에서는 그 하쉬함수를 계산하는데 걸리는 시간이 모든 문자들을 리용하여 계산한 함수보다 약간 작게 된다.

구체적인 프로그람작성에서 중요한것은 충돌을 처리하는것이다. 만일 어떤 요소가 삽입될 때 이미 삽입된 요소와 같은 값으로 하쉬하면 충돌이 일어나게 되는데 이것을 정확히 처리하여야 한다. 충돌을 처리하는데는 여러가지 방법이 있다. 가장 간단한 두가지 **충돌처리**방법들인 사슬주소법과 개방주소지정법을 보게 된다.

제3절. 사슬주소법

일반적으로 **사슬주소법**(*separate chaining*)이라고 하는 첫번째 방법은 같은 값으로 하쉬하는 모든 요소들을 하나의 련결목록에 보관하는것이다. 여기에서는 제3장에서의 목록실현방법을 리용할수 있다. 만일 공간이 충분하지 못하면 그것들을 리용하지 않는것이 좋다(선두매듭들이 기억공간을 랑비하기때문에). 이 절에서는 열쇠단어들이 첫 10개의 완전한 두제곱수들이고 하쉬함수는 간단히 $hash(x)=x \bmod 10$이라고 가정한다. (표크기는 씨수가 아니지만 여기서는 간단히 고찰하기 위하여 그렇게 취급한다.) 그림 5-2는 이것을 명백히 보여 준다.

find를 수행하기 위하여서는 하쉬함수를 리용하여 탐색하려는 목록을 결정한다. 그 다음에 그 목록에서 find를 수행한다. insert를 실행하기 위하여서는 해당 목록을 검사하여 그 세포가 이미전에 배치되여 있는가를 본다. 만일 삽입하려는 요소가 새로운 요소이면 목록의 앞에 삽입하는데 그것은 그 위치가 접근하기 쉽고 또한 제일 마지막에 삽입된

요소들이 먼저 접근될수 있는 경우가 많기때문이다.

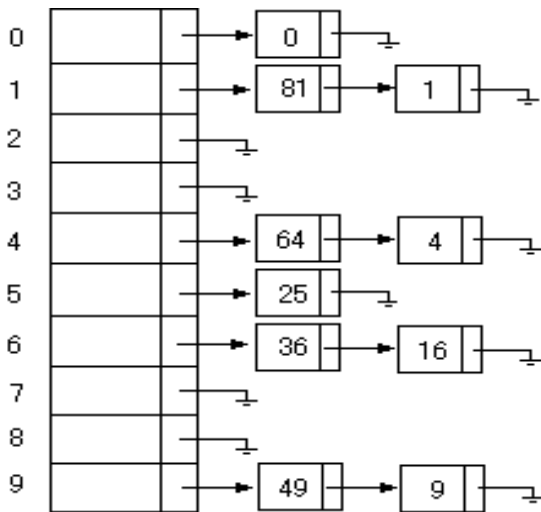



그림 5-2. 사슬주소법에 의한 하쉬표

사슬주소법실현에 대한 클라스대면부를 프로그람 5-4에 보여준다. 하쉬표구조는 련결목록을 가진 배렬로 이루어 지는데 그것은 구축자에서 할당한다.

이 클라스대면부에는 하나의 문법적인 문제가 있다. 즉 the Lists의 선언에서 두개의 > 기호들사이에 하나의 공백이 요구되는데 그것은 >>기호가 C++에서 쓰이는 연산자일수 있고 또한 클라스대면부에 >기호가 두개이상 있을수 있기때문이다.

```
template <c1ass HashedObj>
class HashTable
{
      public:
        explicit HashTab1e( const HashedObj & notFound, int size = 101 );
        HashTab1e( const HashTable & rhs )
              :ITEM_NOT_FOUND( rhs.ITEM_.NOT_.FOUND ), theLists( rhs.theLists ) { }

        const HashedObj & find( const HashedObj & x ) const;

        void makeEmpty( );
        void insert( const HashedObj & x );
        void remove( const HashedObj & x );

        const HashTable & operator=( const HashTable & rhs );

   private:
        vector<List<HashedObj> > theLists;      // The array of Lists
        const HashedObj ITEM_NOT_FOUND;
};
int hash( const string & key, int tableSize );
int hash( int key, int tableSize );
```

프로그람 5-4. 사슬주소하쉬표의 형선언

하쉬표는 또한 2진탐색나무에서 본 ITEM_NOT_FOUND수법과 같은것도 리용한다. 그리고 암시적인 해체자가 쓰일수 있다. 그러나 변하지 않는 자료성원의 리용은 HashTable에 대하여 암시적인 연산자 operator=가 더 이상 필요 없다는것을 의미하는데 그것은 변하지 않는 자료성원이 대입될수 없기때문이다. 그러나 별명검사후에 vector를 복사하기 위하여 operator=를 실현하는것은 간단하며 이것은 직결코드로 수행된다. 복사구축자는 암시적으로 접수될수 있다. 그러나 어떤 번역기들은 변하지 않는 자료성원들을 초기화자표에 명백히 표시할것을 요구하는데 이것이 복사구축자를 명백히 실현하고 제공하여야 할 리유이다(코드량이 작으면 클라스대면부안에서 실현한다.).

2진탐색나무가 Comparable인 객체들에서만 동작하기때문에 이 장에 있는 하쉬표들은 하나의 하쉬함수와 같기연산자(operator==나 operator!=, 또는 가능하면 둘다)들을 제공하는 객체들에 대해서만 동작한다. 이것은 이 하쉬함수들이 HashTable클라스에서 공개함수들로 제공되기때문에 int나 string을 자동적으로 포함하게 된다.

프로그람 5-5는 name성원을 열쇠로 사용하여 일반적인 하쉬표에 보관할수 있는 Employee클라스를 보여 준다. Employee클라스는 같기연산자와 하쉬함수를 제공하는 방법으로 HashedObj요구들을 실현한다.

```
// Example of an Employee class
class Employee
{
   public:
      bool operator==( const Employee & r'hs ) const
           { return name == rhs.name; }
      bool operator!=( const Employee & rhs ) const
           { return ! ( *this == rhs ); }
             // Additional methods and data members

    private:
       string name;
       double salary;
       int       seniority;
          // Additional methods and data members };
 }

 int hash( const Employee & item, int tableSize )
 {
        return hash( item.name, tableSize );
 }
```

프로그람 5-5. hashedobj를 리용할수 있는 클라스의 실례

일부 번역기들에서는 함수형타선언을 하쉬표대면부파일에 추가할 필요가 있다. 즉

```
template <class HashedObj>
int Hash( const HashedObj & x, int tableSize);
```

으로 표시한다. 그러나 다른 번역기들에서는 이것이 여러가지 의미로 쓰이여 int와 string 형들에 대한것들을 쓸수 없게 한다. 일반적으로 우에서 설명된 형타의 형에 대응하는 하쉬함수에 대한 선언이 클라스형타에 먼저 나타나면 함수형타선언을 하지 않을수 있다. 다시말하면 main에서는

```
int Hash(const MyObject & x, int tableSize);
# include "Separate chaining.h"
```

로 시작한다.

프로그람 5-6은 구축자들과 makeEmpty를 보여 준다.

```
/**
 * Construct the hash table.
 */
template <c1ass HashedObj>
HashTab1e<HashedObj>::HashTab1e( const HashedObj & notFound, int size )
      : ITEM_NOT_FOUND( notFound ), theLists( nextPrime( size ) )
{
}

  /**
   * Make the hash table logically empty.
   */
template <c1ass HashedObj>
void HashTab1e<HashedObj>::makeEmpty( )
{
        for( int i = 0; i < theLists.size( ); i++ )
              theLists[ i ].makeEmpty( );
 }
```

프로그람 5-6. 사슬주소법하쉬표에서의 구축자와 makeEmpty연산

find(x)호출은 x에 대응하는 객체에 대한 변하지 않는 참조를 되돌린다. find와 remove를 실현하는 함수를 프로그람 5-7에 보여 준다.

```
/**
 * Remove item x from the hash table.
 */
template <c1ass HashedObj>
void HashTab1e<HashedObj>::remove( const HashedObj & x )
```

```
{
     theLists[ hash( x, theLists.size( ) ) ].remove( x );
}

/**
* Find item x in the hash table.
* Return the matching item, or ITEM_NOT_FOUND, if not found.
  */
template <c1ass HashedObj>
const HashedObj & HashTab1e<HashedObj>: :find( const HashedObj & x ) const
{
      ListItr<HashedObj> itr;
      itr = theLists[ hash( x, theLists.size( ) ) ].find( x );
      return itr.isPastEnd( ) ? ITEM_NOT_FOUND : itr. retrieve( ):
}
```

프로그람 5-7. 사슬주소하쉬표에서의 find와 remove루틴들

다음으로 삽입루틴들을 보자. 삽입되여야 할 항목들이 이미 있으면 아무러한 처리도 하지 않는다. 만일 없으면 그것을 목록의 앞에 배치한다(프로그람 5-8). 그 요소는 목록의 어디에나 배치될수 있는데 우에서와 같은 경우에는 목록의 앞에 배치하는것이 가장 합리적인것으로 된다. whichList는 참조변수인데 이러한 참조변수의 리용에 대해서는 이미 제1장 제5절 4에서 설명되였다.

```
/**
* Insert item x into the hash table. If the item is
* already present, then do nothing.
*/
template <c1ass HashedObj>
void HashTab1e<HashedObj>::insert( const HashedObj & x )
{
   List<HashedObj>,& whichList = theLists[ hash( x, theLists.size( ) ) ];
   ListItr<HashedObj> itr = whichList.find( x );

   if( itr.isPastEnd( ) )
        whichList.insert( x, whichList.zeroth( ) );
}
```

프로그람 5-8. 사슬주소법하쉬표에서의 insert루틴

하쉬루틴들이 삭제연산을 포함하지 않을 때에는 대체로 선두매듭을 쓰지 않는것이 좋은데 그것은 선두매듭들의 리용이 간단하지 않고 적지 않은 공간을 랑비하기때문이다.

어떤 체계는 충돌을 처리하기 위하여 련결목록을 쓰지 않는다. 2진탐색나무나 다른

하쉬표들에서는 련결목록을 쓰지만 표가 크고 하쉬함수가 좋으면 목록들이 모두 짧아 지기때문에 이러한 복잡한 조작을 할 필요가 없다.

하쉬표에 있는 요소들의 수와 하쉬표크기에 대한 비를 계산하여 하쉬표의 부하률 λ를 정의한다. 우의 실례에서는 $\lambda=1.0$이다. 여기에서 λ는 목록의 평균길이이다. 탐색하는데 걸리는 시간은 하쉬함수를 계산하는데 걸리는 상수적인 시간에 목록을 순회하는데 걸리는 시간을 더한 값이다. 탐색이 실패하면 검사하여야 할 매듭의 수는 평균적으로 λ개이다. 성공적인 탐색은 대략 $1+(\lambda/2)$의 련결들이 순회될것을 요구한다. 이것을 보기 위하여서는 탐색되고 있는 목록이 그 쌍을 보관하는 하나의 매듭과 령 또는 그이상의 다른 매듭들을 더한것을 포함한다는것에 주의하여야 한다. N개의 요소들을 가진 표에서《다른 매듭들》의 기대되는 수와 M개의 목록들은 $(N-1)/M=\lambda-1/M$으로 되는데 여기에서 이것은 M이 크게 추정되므로 대체로 λ로 된다. 평균경우에 《다른 매듭들》을 절반정도 탐색하여 그에 일치하는 매듭과 결합시키므로 $1+\lambda/2$개의 매듭들에 대한 평균경우탐색값을 얻는다. 이 분석은 표크기는 실제적으로 중요하지 않지만 부하률은 중요하다는것을 보여 준다. 사슬주소하쉬법의 일반적인 규칙은 하쉬표를 기대되는 요소들의 수(다시말하면 $\lambda\approx1$)만큼 크게 만드는것이다. 앞에서 언급된것처럼 좋은 할당을 담보하기 위하여서는 역시 하쉬표의 크기를 씨수로 하는것이 좋다.

제4절. 개방주소지정법

사슬주소하쉬법은 련결목록을 리용한다는 결함이 있다. 이것은 새로운 세포를 할당하는데 요구되는 시간으로 하여 알고리듬의 속도를 떠지게 하는 경향이 있으며 따라서 본질적으로 두번째 자료구조를 실현할것을 요구한다. **개방주소지정하쉬법**(Open addressing hashing)은 련결목록들을 가지고 충돌을 처리하는 또 하나의 방법이다. 개방주소지정하쉬법체계에서 충돌이 발생하면 빈 세포를 찾을 때까지 다른 세포들을 조사한다. 더 형식적으로 고찰하면 $f(0)=0$을 가지고 $h_i(x)=\text{hash}(x)+f(i))\ \text{mod}\ TableSize$로 계산되는 $h_0(x)$, $h_1(x)$, $h_2(x)$, …의 세포들을 련속적으로 조사한다. 함수 f는 **충돌처리**(Collision resolution)전략이다. 자료들이 모두 표내부에 들어가므로 사슬주소법보다 개방주소지정법을 사용하려면 더 큰 표들이 요구된다. 일반적으로 그 부하률은 개방주소지정법에서 $\lambda=0.5$이하로 떨어 진다. 이제부터 3가지의 일반적인 충돌처리방법들을 보기로 하자.

1. 선형탐색법

선형탐색법에서 f는 일반적으로 $f(i)=i$인 i에 대한 선형함수이다. 이것은 빈세포들에 대한 탐색에서 순차적으로(휘감기표식을 가지고) 시험하는 세포들의 량으로 된다. 표

5-1은 우에서와 동일한 하쉬함수와 충돌처리전략 $f(i)=i$을 리용하여 하쉬표에 열쇠 [89, 18, 49, 58, 69]를 삽입한 결과를 보여 준다.

49가 삽입될 때 첫번째 충돌이 발생되는데 그것은 다음에 리용할수 있는 위치 즉 0 위치에 넣어 지는데 이것은 개방되여 있다. 열쇠 58은 18과 89, 그다음 49와 충돌하며 결국 빈 세포는 셋만큼 떨어 져 나타난다. 69에 대한 충돌은 류사한 방법으로 조종된다. 표가 충분히 클수록 언제나 개방세포를 찾을수 있지만 수행시간은 대단히 커질수 있다. 한층 더 나쁜 상태 즉 가령 표가 상대적으로 비여 있어도 차지된 세포들의 블로크들은 처리를 시작한다. **1차묶음법**(primary clustering)으로 알려 진 이 방법은 그 묶음에 하쉬하는 어떤 열쇠는 충돌을 처리하기 위하여 여러번 시도한 다음에야 묶음에 추가되게 된다는것을 의미한다.

여기에서 더 계산하지 않는다고 하여도 선형탐색법을 리용한 조사수는 삽입들과 실패한 탐색들에서 대략 $\frac{1}{2}(1+1/(1-\lambda)^2)$정도이며 성공한 탐색에서는 $\frac{1}{2}(1+1/(1-\lambda))$이다.

표 5-1. 매 삽입연산후에 선형탐색법을 가지는 개방주소지정하쉬법

	빈 표	89삽입후	18삽입후	49삽입후	58삽입후	69삽입후
0				49	49	49
1					58	58
2						69
3						
4						
5						
6						
7						
8			18	18	18	18
9		89	89	89	89	89

그 계산들은 어느 정도 복잡하다. 그것은 삽입들과 실패한 탐색들이 같은 조사수를 요구하는 코드에서 고찰하는것이 쉽다. 이 사실은 성공한 탐색들이 실패한 탐색들보다 평균적으로 시간이 더 작게 걸린다는것을 보여 준다.

대응하는 식들은 묶음이 문제로 제기되지 않으면 상당히 유도하기 쉽다. 표가 상당히 크고 매개 조사가 앞단계의 조사들에 대하여 독립적이라고 가정하자. 이 가정들은 우연적인 충돌처리전략에 의해 만족되며 λ가 1로 다가가지 않는 한 합리적이다. 우선 탐색이 실패할 때 기대되는 조사수를 구한다. 이것은 빈 세포를 찾을 때까지의 기대되는 조사수이다. 빈 세포들의 작은 부분이 $1-\lambda$이므로 조사하는데 기대되는 세포들의 수는 $1/(1-\lambda)$이다. 성공적인 탐색에 대한 조사수는 개별적인 세포가 삽입될 때 요구되는 조사

수와 같다. 한개 요소가 삽입될 때에는 탐색이 실패할 때의 결과처럼 처리된다. 따라서 성공적인 탐색의 평균값을 계산하기 위하여 탐색이 실패할 때의 값을 리용할수 있다.

주의할 점은 λ가 0에서 그것의 현재값까지 변한다는것이며 따라서 이미전에 삽입들은 더 쉽게 얻어 지고 그 시간은 평균이하로 떨어 진다는것이다. 실례로 우의 표에서 λ=0.5이지만 18이 접근하는 값은 18이 삽입될 때 결정된다. 그때 λ=0.2이므로 18이 상대적으로 빈표에 삽입되였기때문에 그 접근은 69와 같이 마지막에 삽입된 요소가 접근하는 것보다 더 쉬워 진다. 삽입시간의 의미값들을 계산하기 위하여 적분식을 리용하여 평균값을 평가할수 있는데 그 적분식은 다음과 같다.

$$I(\lambda)=\frac{1}{\lambda}\int_0^1\frac{1}{1-x}dx=\frac{1}{\lambda}\ln\frac{1}{1-\lambda}$$

이 식들은 선형탐색에 대응하는 식들보다 명백히 더 좋다. 묶음은 리론적인 문제뿐아니라 실천적인 실현에서도 발생한다.

그림 5-2는 보다 우연적인 충돌처리로 기대되는 값을 가지고 선형탐색(점선곡선들)의 실행을 비교한다. 성공적인 탐색들은 S에 의해서 표시되며 실패한 탐색들과 삽입들은 각각 U와 I로 표시된다.

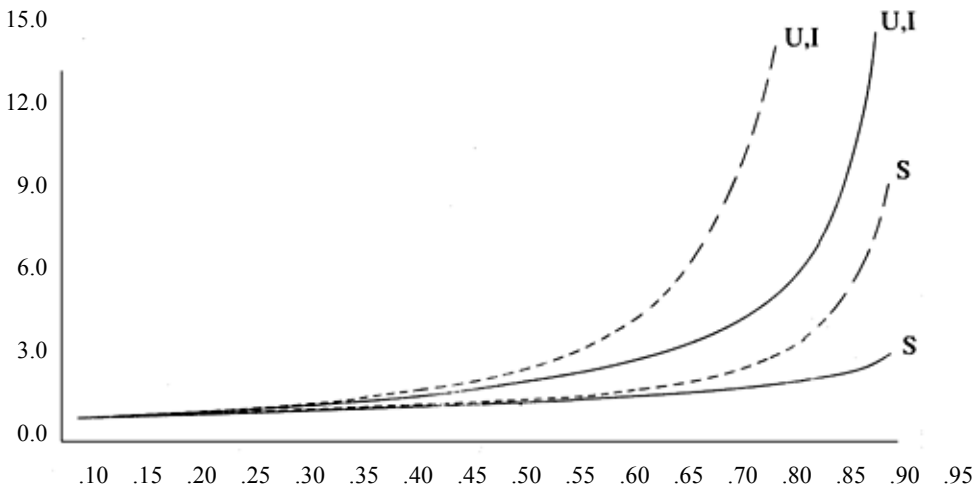



그림 5-3. 선형탐색법(점선)과 우연적인 방법에 대하여 부하률로 표시한 재탐색수들 (S는 성공적인 탐색, U는 탐색실패, I는 삽입)

만일 λ=0.75이면 이 식은 선형탐색법에서 삽입할 때 8.5회의 재탐색들이 요구된다는것을 의미한다. λ=0.9이면 50회의 재탐색들이 요구되는데 이것은 의의가 없다. 이것은 묶음의 크기가 문제로 되지 않았을 때에는 각각 4번과 10번의 재탐색으로 매개 부하률들을 비교한다. 그러나 이 식들로부터 표가 절반이상 차지되게 될 때는 선형탐색이 좋지

묶음법은 극히 적은 리론적인 결함을 가지고 있다. 모의결과는 일반적으로 2차묶음법이 매번 탐색할 때마다 절반이하를 조사한다는것을 보여 준다. 다음의 방법은 이것을 배제하지만 추가적인 곱하기와 나누기를 가진다.

3. 2중하쉬법

마지막충돌처리산법은 **2중하쉬법**(double hasing)이다. 2중하쉬법에서 하나의 일반적인 선택은 $f(i)=i\cdot hash_2(x)$이다. 이 식은 두번째 하쉬함수를 x에 적용하고 $hash_2(x)$, $2hash_2(x)$, … 의 거리에 대하여 재탐색한다는것을 의미한다. $hash_2(x)$의 불충분한 선택은 대단히 나쁜 결과를 준다. 실례로 명백한 선택 $hash_2(x)=x$ mod 9는 99가 앞의 실례들에서 입력에 삽입되였을 때에는 효과가 없다. 따라서 그 함수는 결코 0으로 평가하지 말아야 한다. 또한 모든 세포들이 재탐색될수 있다는것을 명백히 하는것도 중요하다(표크기가 씨수가 아니기때문에 이것은 아래의 실례에서는 불가능하다.). *TableSize*보다 더 작은 씨수 R를 가진 $hash_2(x)=R-(x \bmod R)$와 같은 함수는 잘 처리된다. R=7을 선택하면 그때 표 5-3은 앞에서와 같은 열쇠단어들을 입구하는 결과를 보여 준다.

표 5-3. 매개 삽입후에 2중하쉬를 가진 개방주소지정하쉬표

	빈표	**89삽입후**	**18삽입후**	**49삽입후**	**58삽입후**	**69삽입후**
0						69
1						
2						
3					58	58
4						
5						
6				49	49	49
7						
8			18	18	18	18
9		89	89	89	89	89

첫번째 충돌은 49가 삽입될 때 발생한다. $hash_2(49)=7-0=7$이고 따라서 49는 위치 6에 삽입된다. $hash_2(58)=7-2=5$이므로 58은 위치 3에 삽입된다. 마지막으로 69가 충돌하면 거리 $hash_2(69)=7-6=1$만큼 떨어 져서 삽입된다. 만일 위치 0에 60을 삽입하려고 한다면 충돌이 발생한다. $hash_2(60)=7-4=3$이므로 빈 위치가 나타날 때까지 3, 6, 9, 2위치들에서 진행되게 된다. 그것은 일반적으로 일부 좋지 못한 경우에 탐색하는것은 가능하지만 여기에서는 너무 많아서 찾을수 없다.

정리 5-1.

만일 2차탐색법에서 표크기가 씨수이고 표가 적어도 절반 비여 있다면 언제나 새로운 요소를 삽입할수 있다.

증명:

표크기 *TableSize*가 3보다 더 큰 씨수(기수)라고 하자. 첫 $\lceil TableSize/2 \rceil$개의 세포들에 대한 위치들(초기위치 $h_0(x)$를 포함하는)은 모두 명백하다. 이것들가운데서 2개의 위치는 $h(x)+i^2 \pmod{TableSize}$와 $h(x)+j^2 \pmod{TableSize}$인데 여기서 $0 \leqq i,\ j \leqq \lfloor TableSize/2 \rfloor$이다. 량립될수 없는 리유로 하여 이 위치들은 갈지만 $i \neq j$라고 가정하자. 그때

$$
\begin{aligned}
h(x)+i^2 &= h(x)+j^2 && \pmod{TableSize}\\
i^2 &= j^2 && \pmod{TableSize}\\
i^2 - j^2 &= 0 && \pmod{TableSize}\\
(i-j)(i+j) &= 0 && \pmod{TableSize}
\end{aligned}
$$

*TableSize*가 씨수이므로 $(i-j)$나 $(i+j)$가 $0 \pmod{TableSize}$과 같게 된다. i와 j가 서로 다른 위치이므로 첫번째의 선택은 불가능하다. $0 \leqq i,\ j \leqq \lfloor TableSize/2 \rfloor$이므로 두번째의 선택도 불가능하다. 따라서 첫번째 $\lceil TableSize/2 \rceil$대신의 위치들이 명백히 구별된다. 만일 최대로 $\lfloor TableSize/2 \rfloor$개 위치들이 주어 지면 빈 위치는 언제나 찾을수 있다.

만일 빈세포가 표크기의 절반보다 하나 더 작다고 해도 삽입은 실패한다(이것이 극히 류사하지 않다고 하더라도). 따라서 이것을 명심하는것이 중요하다. 또한 표크기를 씨수로 정하는것이 중요하다.[17] 만일 표크기가 씨수가 아니면 선택적인 위치들의 수는 심하게 줄어 든다. 실례로 만일 표크기가 16이면 선택적인 위치들은 오직 1이나 4 또는 9만큼 떨어 진 거리에 있게 된다.

개방주소지정하쉬표에서는 표준적인 삭제를 진행할수 없는데 그것은 그 세포가 그 위치를 지나가도록 충돌을 발생시키기때문이다. 실례로 89를 삭제하면 남아있는 find연산들이 대체로 모두 실패한다. 따라서 개방주소지정하쉬표들은 이 경우에 실제로 퇴화되지 않는다고 하더라도 지연삭제를 요구한다.

이 클라스대면부는 프로그람 5-9에서 보여준 개방주소지정하쉬법을 실현하는데 요구된다. 그리고 목록들의 배렬대신에 하쉬표입구점세포들에 대한 배렬을 리용한다.

---

[17] 만일 표크기가 $4k+3$ 형태의 소수이고 2차충돌처리전략의 $f(i)=\pm i^2$이 리용되면 전체 표가 재탐색될수 있다. 그 값은 좀 더 복잡한 루틴이다.

계층적인 클라스 HashEntry는 info성원에 있는 입구점의 상태를 보관하는데 이 상태는 ACTIVE이거나 EMPTY 또는 DELETED이다.

```
template <c1ass HashedObj>
class HashTable
{
    public:
        explicit HashTable( const HashedObj & notFound, int size = 101 );
        HashTab1e( const HashTable & rhs ) :currentSize( rhs.currentSize ),
        ITEM_NOT_FOUND( rhs.ITEM_NOT_FOUND ), array( rhs.array ) { }

        const HashedObj, & find( const HashedObj & x ) const;

        void makeEmpty( );
        void insert( const HashedObj & x );
        void remove( const HashedObj & x );

        const HashTable & operator=( const HashTable & rhs );

        enum EntryType { ACTIVE, EMPTY, DELETED };

    private:
        struct HashEntry
        {
           HashedObj element;
           EntryType info;

           HashEntry( const HashedObj & e = HashedObj( ), EntryType i = EMPTY )
                : etement( e ), info( i ) { }
        };

        vector<HashEntry> array;
        int currentSize;
        const HashedObj ITEM_NOT_FOUND;

        bool isActive( int currentPos ) const;
        int findPos( const HashedObj & x ) const;
        void rehash( );
};
```

프로그람 5-9. 계층적인 HashEntry클라스를 포함하는 개방주소지정하쉬표들에 대한 클라스대면부

C++에서 이 상수들은 초기값을 가진 static const자료성원들로써 선언될수 있다. 따라서 HashTable클라스에서

static const int ACTIVE=0;

static const int EMPTY=1;
static const int DELETED=2;

를 가지게 된다. 그러나 이것은 ANSI표준으로 규정되였음에도 불구하고 모든 번역기들은 이것을 제공하지 않는다. 따라서 렬거형을 리용하는데 즉

enum Entry Type {ACTIVE, EMPTY, DELETED};

은 같은 결과를 나타낸다. EntryType형은 어떤 int보다 더 크지 않으며 ACTIVE와 EMPTY, DELETED의 값들은 순차적으로 번역기에 대입된다. 이 수법은 C++프로그람작성자들에 의해 옹근수클라스상수들을 선언하는데 널리 리용된다.

실례로 다음과 같은 선언을 할수 있다.

enum {MAX_VALUE=10};

표의 구축(프로그람 5-10)은 매개 세포에 대하여 info성원을 EMPTY로 설정한다.

```
/**
 * Construct the hash table.
 */
template <c1ass HashedObj>
HashTab1e<HashedObj>::HashTab1e( const HashedObj & notFound, int size )
  : ITEM_NOT_FOUND( notFound ), array( nextPrime( size ) )
{
    makeEmpty( );
}

/**
 * Make the hash table logically empty,
 */
template <c1ass HashedObj>
void HashTab1e<HashedObj>::makeEmpty( )
{
     currentSize = 0;
     for( int i = 0; i < array. size( ); i++ )
          array[ i ].info = EMPTY;
}
```

프로그람 5-10. 개방주소지정하쉬표를 초기화하는 루틴

사슬주소하쉬법에서처럼 프로그람 5-11에서 보여 주는 find(x)는 하쉬표에서 x와 일치하는 객체에 대한 참조를 되돌린다. 만일 x가 없으면 find는 ITEM_NOT_FOUND를 되돌린다. private성원함수 findPos는 충돌처리를 실행한다. 하쉬표가 적어도 표에 있는 요소들의 2배만큼 크다고 하면 2차탐색처리는 언제나 수행된다. 만일 그렇지 않으면 4행전에서 *i*(collisionNum)를 검사할 필요가 있다. 프로그람 5-11의 실현에서 삭제하기 위하여

표시된 요소들은 그 표에 존재하는것으로 계산한다. 이것은 여러가지 문제들을 발생시키는데 그것은 표가 너무 때이르게 충만될수 있기때문이다. 이 내용을 구체적으로 보자.

```
         /**
          * Find item x in the hash table.
          * Return the matching item or ITEM_NOT_FOUND if not found.
          */
         template <c1ass HashedObj>
         const HashedObj & HashTab1e<HashedObj>::find( const HashedObj & x ) const
         {
                int currentPos = findPos( x );
                return isActive( currentPos ) ? array[ currentPos ].element
                                                      : ITEM_NOT_FOUND;
         }
         /**
           * Method that performs quadratic probing resolution.
           * Return the position where the search for x terminates.
           */
         template <c1ass HashedObj>
         int HashTab1e<HashedObj>: :findPos( const HashedObj & x ) const
         {
/*1*/         int collisionNum = 0;
/*2*/         int currentPos = hash( x, array.size( ) );

/*3*/         whi1e( array[ currentPos ].info != EMPTY &&
                     array[ currentPos ].element != x )
              {
/*4*/            currentPos += 2 * ++collisionNum - 1;     // Compute ith probe
/*5*/            if( currentPos >= array.size( ) )
/*6*/                currentPos -= array.size( );
               }
/*7*/          return currentPos;
          }
          /**
.          * Return true if currentPos exists and is active
           */
         template <c1ass HashedObj>
         bool HashTab1e<HashedObj::isActive( int currentPos ) const
         {
              return array[ currentPos ].info == ACTIVE;
         }
```

프로그람 5-11. 2차탐색하쉬법에 대한 find루틴과 private 성원함수

4행부터 6행까지는 2차탐색처리를 수행하는 고속방법을 표현한다. 2차탐색처리함수의 정의로부터 $f(i)=f(i-1)+2i-1$이며 따라서 처리할 다음세포는 2를 곱하고(실제로는 비트밀

기로써) 1만큼 덜어서 결정할수 있다. 새로운 위치가 배렬의 크기를 넘어 서면 그것을 *TableSize*로 덜어서 그 범위안에 다시 넣을수 있다. 이것은 곱하기와 나누기를 쓰지 않기 때문에 명백히 그 산법보다 더 빠르다. 중요하게 주의할 문제는 3행에서 검사순서이다. 그것을 교체하지 말아야 한다.

마지막루틴은 삽입이다. 사슬주소하쉬법에서와 같이 x가 이미 존재하면 아무러한 처리도 하지 않는다. 그 대신 간단히 수정하여 다른것을 처리한다. x가 없으면 findPos루틴에 의해서 제공된 위치에 배치한다. 이 코드는 프로그람 5-12에서 보여 준다.

```
/**
 * Insert item x into the hash table. If the item is
 * already present, then do nothing.
 */
template <c1ass HashedObj>
void HashTab1e<HashedObj>::insert( const HashedObj & x )
{
        // Insert x as active
    int currentPos = findPos( x );
    if( isActive( currentPos ) )
        return;
    array[ currentPos ] = HashEntry( x, ACTIVE );
        // Rehash; see Section 5.5
    if( ++currentSize > array.size( ) / 2 )
        rehash( );
}

/**
* Remove item x from the hash table.
*/
template <c1ass HashedObj>
void HashTab1e<HashedObj>: :remove( const HashedObj & x )
{
    int currentPos = findPos( x );
    if( isActive( currentPos ) )
        array[ currentPos ].info = DELETED;
}
```

프로그람 5-12. 2 차탐색법을 리용한 하쉬표의 insert 루틴

부하률이 0.5를 초과하면 표는 충만상태이다. 이때 례외를 발생시킬수도 있지만 그 대신에 하쉬표를 확대하는 방안을 리용한다. 이것을 재하쉬법이라고 하며 제5장 제5절에서 설명한다.

2차탐색이 1차묶음법을 배제한다고 하더라도 같은 위치에 하쉬하는 요소들은 같은 선택적인 세포들을 조사할것이다. 이것을 **2차묶음법**(secondary clustering)이라고 한다. 2차

묶음법은 극히 적은 리론적인 결함을 가지고 있다. 모의결과는 일반적으로 2차묶음법이 매번 탐색할 때마다 절반이하를 조사한다는것을 보여 준다. 다음의 방법은 이것을 배제하지만 추가적인 곱하기와 나누기를 가진다.

3. 2중하쉬법

마지막충돌처리산법은 **2중하쉬법**(double hasing)이다. 2중하쉬법에서 하나의 일반적인 선택은 $f(i)=i \cdot hash_2(x)$이다. 이 식은 두번째 하쉬함수를 x에 적용하고 $hash_2(x)$, $2hash_2(x)$, … 의 거리에 대하여 재탐색한다는것을 의미한다. $hash_2(x)$의 불충분한 선택은 대단히 나쁜 결과를 준다. 실례로 명백한 선택 $hash_2(x)=x$ mod 9는 99가 앞의 실례들에서 입력에 삽입되였을 때에는 효과가 없다. 따라서 그 함수는 결코 0으로 평가하지 말아야 한다. 또한 모든 세포들이 재탐색될수 있다는것을 명백히 하는것도 중요하다(표크기가 씨수가 아니기때문에 이것은 아래의 실례에서는 불가능하다.). *TableSize*보다 더 작은 씨수 R를 가진 $hash_2(x)=R-(x \bmod R)$와 같은 함수는 잘 처리된다. R=7을 선택하면 그때 표 5-3은 앞에서와 같은 열쇠단어들을 입구하는 결과를 보여 준다.

표 5-3. 매개 삽입후에 2중하쉬를 가진 개방주소지정하쉬표

	빈표	89 삽입후	18 삽입후	49삽입후	58삽입후	69 삽입후
0						69
1						
2						
3					58	58
4						
5						
6				49	49	49
7						
8			18	18	18	18
9		89	89	89	89	89

첫번째 충돌은 49가 삽입될 때 발생한다. $hash_2(49)$=7−0=7이고 따라서 49는 위치 6에 삽입된다. $hash_2(58)$=7−2=5이므로 58은 위치 3에 삽입된다. 마지막으로 69가 충돌하면 거리 $hash_2(69)$=7−6=1만큼 떨어 져서 삽입된다. 만일 위치 0에 60을 삽입하려고 한다면 충돌이 발생한다. $hash_2(60)$=7−4=3이므로 빈 위치가 나타날 때까지 3, 6, 9, 2위치들에서 진행되게 된다. 그것은 일반적으로 일부 좋지 못한 경우에 탐색하는것은 가능하지만 여기에서는 너무 많아서 찾을수 없다.

앞에서 언급한것처럼 우에서 리용한 간단한 하쉬표의 크기는 씨수가 아니다. 하쉬함수를 편리하게 계산하기 위하여 이것을 리용하였지만 2중하쉬법을 리용할 때에는 표의 크기를 정확히 씨수로 취급하는것이 중요하다. 만일 표에 23을 삽입하려고 하면 그것은 58과 충돌한다. $hash_2(23)=7-2=5$이고 표크기가 10이기때문에 본질적으로 하나의 선택적인 위치만을 가지는데 그것은 이미 차지되였다. 따라서 표의 크기가 씨수가 아니면 너무 때이르게 선택적인 위치에서 벗어 나게 된다. 그러나 2중하쉬법이 정확히 실현되면 모의들은 기대되는 재탐색들의 수가 우연적인 충돌처리전략과 거의 같아 진다는것을 보여 준다. 이것은 2중하쉬법이 리론적으로 아주 흥미있다는것을 보여 준다. 그러나 2차탐색법은 두번째 하쉬함수를 요구하지 않으므로 따라서 실천적으로 더 간단하고 더 빠르다.

제5절. 재하쉬법

만일 표가 너무 충만되면 연산들의 실행시간이 대단히 길어 지게 되며 삽입들은 2차탐색처리를 가진 개방주소지정하쉬법에서 실패하게 된다. 이것은 삽입과 혼합된 삭제들이 너무 많을 때 발생할수 있다. 그때의 해결방안은 대략 2배의 크기를 가지는 또 하나의 하쉬표를 구축하고(새로운 하쉬함수로 련합되여) 초기하쉬표전체를 쭉 조사하며 매개요소(삭제되지 않은)에 대한 새로운 하쉬값을 계산하고 그것을 새로운 표에 삽입하는것이다.

실례로 요소 13, 15, 24, 6들이 크기가 7인 개방주소지정하쉬표에 삽입된다고 하자. 하쉬함수는 $h(x)=x \bmod 7$이다. 충돌을 처리하기 위하여 선형탐색법을 리용한다고 하자. 결과적인 하쉬표를 그림 5-4에 주었다.

만일 23이 표에 삽입되면 그림 5-5에 있는 결과적인 표는 70%이상 충만될것이다.

0	6
1	15
2	
3	24
4	
5	
6	13

그림 5-4. 입구 13, 15, 6, 24를 가지고 선형탐색을 리용한 개방주소지정하쉬표

0	6
1	15
2	23
3	24
4	
5	
6	13

그림 5-5. 23이 삽입된후에 선형탐색법을 리용한 개방주소지정하쉬표

표가 너무 충만되기때문에 새로운 표를 만든다. 새로운 표의 크기는 17인데 그것은 17이

처음 표크기의 2배이상 되는 수들가운데서 첫번째 씨수이기때문이다. 새로운 하쉬함수는 그때 $h(x)=x \bmod 17$로 된다. 초기의 표가 조사되고 요소 6, 15, 23, 24, 13들이 새로운 표에 삽입된다. 그 결과표를 그림 5-6에 보여 준다.

0	
1	
2	
3	
4	
5	
6	6
7	23
8	24
9	
10	
11	
12	
13	13
14	
15	15
16	

그림 5-6. 재하쉬후에 개방주소지정하쉬표

이러한 방법을 재하쉬법(rehashing)이라고 한다. 이것은 비용이 많이 드는 연산인데 그 실행시간은 재하쉬하려는 N개의 요소들이 있고 표의 크기가 대체로 $2N$이므로 $O(N)$으로 되지만 실제로는 드물게 나타나기때문에 다 나쁘다고는 할수 없다. 특히 거기에는 마지막 재하쉬에 앞서 $N/2$번의 삽입을 진행하여야 하는데 따라서 실행시간은 본질적으로 매개 삽입에 대하여 상수값만큼 추가된다.[18] 만일 이 자료구조가 프로그람의 한부분으로서 처리되면 그 결과는 잘 알려 지지 않는다. 바꾸어 말하면 하쉬법이 대화형체계의 한 부분으로서 실행되면 그때 삽입속도가 떠지는것을 볼수 있다.

재하쉬법은 2차탐색법을 리용하여 여러가지 방법으로 실현할수 있다. 한가지 방안은 표의 절반이 충만되자마자 재하쉬하는것이다. 다른 방안은 삽입이 실패할 때에만 재하쉬하는것이다. 세번째 방법은 중간처리법(middle-of-the-road)인데 표가 일정한 부하률에 도달할 때 재하쉬하는것이다. 부하률증가에 의하여 실행속도가 떨어지기때문에 적합한 차단을 가지고 실행되는 세번째 방법이 좋다.

재하쉬법은 표크기에 대하여 프로그람작성자가 주의하지 않아도 되게 하며 하쉬표들이 복잡한 프로그람들에서 임의의 크기로 처리될수 있기때문에 중요하다. 련습들에서는 지연삭제를 가지고 재하쉬의 리용을 공동으로 연구할수 있는 문제들을 주었다. 재하쉬법은 또한 다른 자료구조들에서도 리용될수 있다. 실례로 제3장의 대기렬자료구조가 충만되면 2배크기의 배렬을 선언하고 처음의것을 해방하면서 모든 처리를 복사할수 있다.

프로그람 5-13은 재하쉬를 간단히 실현할수 있다는것을 보여 준다.

[18] 이것이 새로운 표가 처음표의 2배만한 크기로 되는 리유이다.

```
/**
 * Expand the hash table.
 */
template <c1ass HashedObj>
void HashTab1e<HashedObj>::rehash( )
{
    vector<HashEntry> oldArray = array;
            // Create new double-sized, empty table
        array.resize( nextPrime( 2 * o1dArray.size( ) ) );
        for( int j = 0; j < array.size( ); j++ )
            array[ j ].info = EMPTY;
            // Copy table over
        currentSize = 0;
        for( int i = 0; i < o1dArray.size( ); i++ )
          if( o1dArray[ i ].info == ACTIVE )
              insert( o1dArray[ i ].element );
}
```

프로그람 5-13. 개방주소지정하쉬표에 대한 재하쉬법

제6절. 확장하쉬법

이 장에서 마지막문제는 자료량이 너무 커서 주기억기에 넣을수 없는 경우에 대한 처리이다. 제4장에서 본것처럼 이에 대한 중요한 개념은 자료를 검색하는데 요구되는 디스크접근수이다.

앞에서와 같이 어떤 문제에서 N개의 기록들을 보관하여야 한다고 하자. 즉 N값이 시간에 따라 변한다고 하자. 더우기 하나의 디스크블로크에는 많아서 M개의 기록들이 넣어 진다. 이 절에서는 M=4를 리용한다.

만일 개방주소지정하쉬법이나 사슬주소하쉬법이 리용되면 거기에서 중요한 문제는 잘 작성된 하쉬표에서도 find연산을 실행할 때 조사하여야 할 여러 블로크들에서 충돌이 발생한다는것이다. 더우기 표가 충만되였을 때 극단한 비용이 드는 재하쉬단계가 실행되여야 하는데 이것은 $O(N)$의 디스크접근을 요구한다.

확장하쉬법으로 알려 진 방법은 find연산을 두개의 디스크접근으로 실행되도록 하는 것이다. 삽입은 또한 적은 수의 디스크접근을 요구한다.

B-나무가 $O(\log_{M/2}N)$의 깊이를 가진다는것을 제4장에서 이미 고찰하였다. M이 증가하면 B-나무의 깊이는 감소된다. 리론적으로는 B-나무의 깊이가 1이 되도록 M을 선택한다. 그때 첫번째 다음의 임의의 find는 한번의 디스크접근을 가지게 되는데 그것은 정확히 뿌리매듭이 주기억기에 보관되기때문이다. 이 방법에서 문제로 되는것은 분기결수가 너무 커서 자료가 있는 잎매듭을 결정하기 위한 어떤 처리를 해야 한다는것이다. 이 단계

의 실행시간이 감소되면 그것은 실천적인 체계로 된다. 이것은 정확히 확장하쉬법에 의해서 리용되는 전략이다.

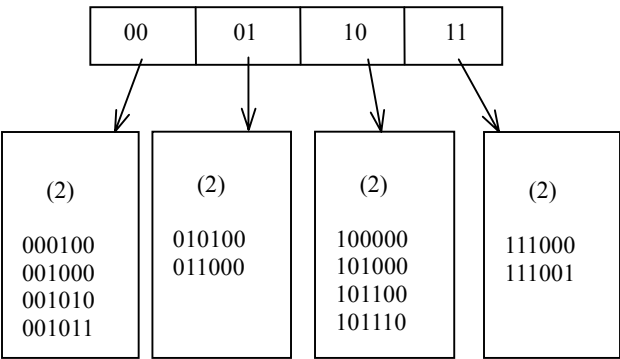



그림 5-7. 확장하쉬법: 초기자료

이 경우에 자료가 여러개의 6bit의 옹근수들로 되여 있다고 하자. 그림 5-7은 이러한 자료들에 대한 확장하쉬법을 보여 준다. 《나무》의 뿌리는 2개비트의 자료에 의해서 결정되는 4개의 지시자들을 포함한다. 매개 잎은 M=4개이하의 세포들을 가진다. 매개 잎에서 첫 두개의 비트들은 완전히 같은데 이것은 괄호안에 있는 수자에 의해 지시된다. 더 형식적으로 고찰하기 위하여 D는 뿌리에 의해서 리용되는 비트들의 수를 나타내는데 이것을 때때로 등록부(directory)라고 한다. 따라서 등록부에서 입구점들의 수는 2^D이다. d_L은 개개의 잎들과 $d_L \leq D$에 의존하게 된다.

이제 열쇠 100100를 삽입한다고 하자. 이것은 세번째 잎으로 선택되여야 하지만 세번째 잎은 이때 충만상태이므로 거기에는 삽입할 자리가 없다. 따라서 이 잎을 두개의 잎으로 가르는데 이것은 현재 첫 세개의 비트들에 의해서 결정된다. 이것은 등록부크기를 3비트로 증가시킬것을 요구한다. 이것을 그림 5-8에 보여 주었다.

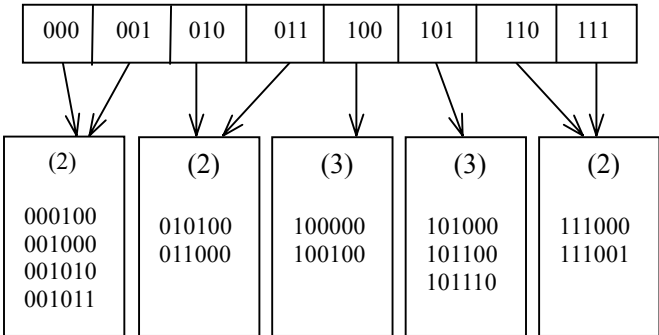



그림 5-8. 확장하쉬법: 100100을 삽입하고 등록부를 가른 다음

매듭가르기에 참가하지 않는 모든 잎들은 현재 두개의 린접등록부입구점에 의해 표시된다. 따라서 전체 등록부가 다시 표시되지만 다른 잎들은 실제적으로 접근되지 않는다.

이제 열쇠 000000이 삽입되면 그때 첫번째 잎이 갈라 지는데 d_L=3인 두개의 잎들이 발생한다. D=3이므로 등록부에서 유일한 변경은 000과 001지시자들을 수정하는것이다 (그림 5-9).

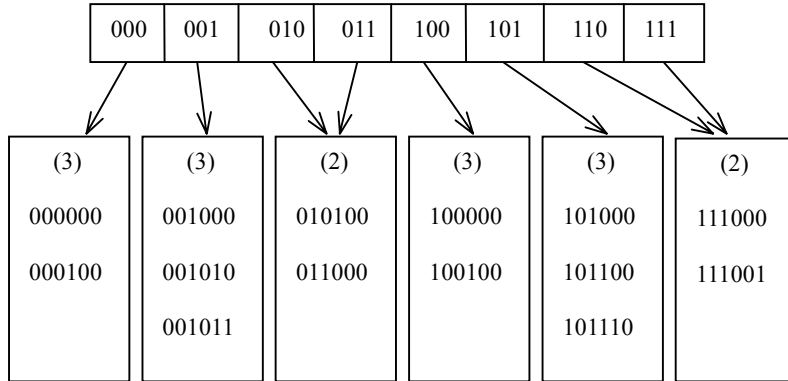



그림 5-9. 확장하쉬법 000000을 삽입하고 잎을 가른 다음

이러한 아주 간단한 방법은 큰 자료기지들에서의 insert와 find연산들에 고속접근시간을 제공한다. 여기에 아직 고찰하지 않는 중요한 문제들이 있다.

첫째로, 잎들에 있는 요소들이 D+1개의 유도비트들보다 더 많게 되면 여러개의 등록부들이 갈라 지게 된다. 실례로 D=2로 처음의 실례에서 시작해 보면 111010과 111011, 그리고 마지막으로 111100이 삽입될 때 등록부크기는 5개의 열쇠들을 구분하기 위하여 4로 증가되여야 한다. 이것은 주의하여야 할 문제이다. 둘째로, 반복되는 열쇠들이 있을수 있는데 거기에 M개이상의 복제가 존재하면 이 알고리듬은 모든 처리를 진행하지 않는다. 이러한 경우에는 다른 구조를 만들어야 한다.

이 가능성들은 그 비트들이 완전히 우연적으로 되여야 한다는것을 암시한다. 이것은 열쇠들을 적당한 크기의 옹근수로 하쉬하는것으로써 실현할수 있다. 이로부터 그 이름을 확장하쉬법이라고 하는것이다.

아주 어려운 분석끝에 얻어 지는 확장하쉬법의 몇가지 실행특성을 언급하는것으로 이 내용을 끝내기로 한다. 이 결과들은 비트패턴들이 같은 형식으로 서술된다는 론리적인 가정에 기초하고 있다.

기대되는 잎들의 수는 $(N/M)\log_2 e$이다. 따라서 평균경우에 잎은 ln2=0.69로 충만된다. 이것은 B-나무에서와 같은것으로써 전체적으로 볼 때 그리 놀랍지 않은것이다. 그것은 두 자료구조에서 새로운 매듭들은 $(M+1)$번째의 입구점이 추가될 때 만들어 지기때문이다. 더 놀라운 결과는 기대되는 등록부의 크기(바꾸어 말하면 2^D)가 $O(N^{1+1/M}/M)$이라는것이다. 만일 M이 대단히 작으면 그 등록부는 지나치게 커질수 있다. 이 경우에는 실제적

인 기록들 대신에 그 기록들에 대한 지시자들을 포함하는 잎들을 가질수 있는데 이때 M의 값은 증가한다. 이것은 더 작은 등록부를 유지하기 위하여 매개 find연산에 두번째 디스크접근을 추가한다. 등록부가 너무 커서 주기억기에 넣을수 없으면 두번째 디스크접근이 필요하게 된다.

요약

하쉬표는 상수적인 평균시간에 insert와 find연산들을 실현하기 위하여 리용될수 있다. 하쉬표들을 리용할 때 부하률과 같은 구체적인 문제들에 관심을 두는것은 특별히 중요하다. 만일 그렇지 않으면 시간한계들이 가치가 없기때문이다. 또한 열쇠가 짧은 문자렬이거나 옹근수가 아닐 때에는 적당한 하쉬함수를 선택하는것이 중요하다.

사슬주소하쉬법에서 그 부하률이 그리 크지 않으면 실행시간은 크게 떨어 지지 않지만 부하률은 1가까이에 있게 된다. 개방주소지정하쉬법에서 피할수 없는 경우를 제외하고 부하률은 0.5를 초과하지 않는다. 만일 선형탐색이 리용되면 부하률이 1에 접근할 때 실행시간은 급격히 떨어 진다. 재하쉬법은 표를 증가(감소)시키려고 할 때 실현될수 있으며 따라서 적당한 부하률을 유지할수 있다. 이것은 기억공간이 부족할 때 방대한 크기의 하쉬표를 선언하는데서는 가능하지 못하다.

또한 insert와 find연산들을 실현하는데 2진탐색나무들을 리용할수 있다. 결과적으로 평균시간한계는 $O(\log N)$이지만 2진탐색나무들은 순서를 요구하는 루틴들을 제공하므로 더 유력하다. 하쉬표를 리용하여 가장 작은 요소를 찾는것은 불가능하다. 또한 문자렬이 정확하지 않으면 문자렬들을 효과적으로 탐색할수 없다. 2진탐색나무는 적당한 범위에 있는 모든 항목들을 고속으로 탐색할수 있는데 이러한 방법을 하쉬표에서는 제공하지 못한다. 더우기 $O(1)$보다 훨씬 더 큰 $O(\log N)$은 필요가 없는데 그것은 특히 2진탐색나무들이 곱하기와 나누기를 요구하지 않기때문이다.

다시말하여 일반적으로 정렬된 입구자료들은 불충분하게 구성되는 2진나무를 만들수 있지만 하쉬법에서는 최악의 경우 실행오유에 귀착된다. 평형탐색나무들은 그것을 실현하는데 비용이 아주 많이 들기때문에 순서정보가 요구되지 않고 또한 입구가 정렬되여 있다는 어떠한 기미만 보이면 하쉬법을 자료구조로 선택하는것이 더 좋다.

하쉬법에 대한 응용들은 수없이 많다. 번역기들은 하쉬표를 리용하여 원천코드에서 선언된 변수들의 자리길을 유지한다. 이 자료구조를 기호표(symbol table)라고 한다. 하쉬표들에서는 오직 insert와 find만 실행되기때문에 이러한 문제들에 대하여 리상적인 응용으로 된다. 식별자들은 형태적으로 짧기때문에 하쉬함수는 이것을 고속으로 계산할수 있다.

하쉬표는 매듭들이 번호대신에 실제이름을 가지는 임의의 그라프리론문제에서도 쓸모가 있다. 여기서 입력자료가 읽어 지면 정점들은 읽어 지는 순서로 1부터 번호를 붙인다. 다시말하면 그 입력자료는 자모순으로 된 입구점들에 대한 큰 모임과 류사하다. 실례로 정점들이 콤퓨터들이라고 하자. 그때 콤퓨터들에 대한 특별한 설치표가 *ibm*1, *ibm*2,

*ibm*3, …으로써 그 콤퓨터들을 표시하면 탐색나무가 리용될 때 극적인 효과를 가지게 된다.

하쉬표에 대한 세번째로 일반적인 리용은 유희프로그람에 있다. 프로그람은 유희의 각이한 방향들을 탐색할 때 그 위치(그리고 그 위치에서 그의 이동을 보관하는)에 기초한 하쉬함수를 계산하여 위치들의 자리길을 유지한다. 이동들의 간단한 호환으로 하여 같은 위치가 다시 발생하면 프로그람은 많은 비용이 드는 재계산을 피할수 있다. 모든 유희프로그람들의 일반적인 특징은 변환표(transposition table)이다.

또 하나의 하쉬법의 리용은 직결맞춤법검사기(on-line spelling checker)들에 있다. 틀린 맞춤법삭제(정확한것과 대조하는 방법으로)가 중요하게 제기되면 전체 사전은 미리 하쉬되여 단어들을 상수시간내에 검사할수 있다. 하쉬표들에서는 단어들을 자모순으로 정렬할 필요가 없기때문에 이러한 문제들에 아주 적당한데 틀린 맞춤법들을 문서에서 발생된 순서로 출력할수 있다.

제1장의 단어맞추기문제를 다시 한번 고찰하는것으로 이 장을 마치자. 제1장에서 서술된 두번째 알고리듬을 리용하고 가장 큰 단어의 크기가 어떠한 작은 상수이면 그때 W개의 단어들을 사전에서 읽어 들이고 그것을 하쉬표에 넣는 시간은 $O(W)$이다. 이 시간은 디스크입출력에 의해서 결정되지 하쉬루틴들에 의해서 결정되지는 않는다. 그 알고리듬의 나머지부분은 매개 순서붙은 4원묶음표(행, 렬, 방향, 문자수)에 대한 하나의 단어표현을 검사한다. 매개 표의 탐색이 $O(1)$이고 8개의 방향과 매 단어당 문자개수만이 있기때문에 이 단계에서의 실행시간은 $O(R, C)$이다. 전체 실행시간은 $O(R, C+W)$인데 이것은 처음의 $O(R, C \cdot W)$보다 명백하게 개선된것이다. 실천적으로 그 실행시간을 감소시키기 위하여 좀 더 최량화할수 있는데 그에 대해서는 련습문제로 제시한다.

련습문제

5-1. 입력자료 {4371, 1323, 6173, 4199, 4344, 9679, 1989}와 하쉬함수 $h(x)=x(\bmod 10)$을 주고 결과를 표시하시오.

ㄱ. 사슬주소하쉬표

ㄴ. 선형탐색법을 리용한 개방주소지정하쉬표

ㄷ. 2차탐색을 리용한 개방주소지정하쉬표

ㄹ. 두번째 하쉬함수 $h_2(x)=7-(x \bmod 7)$을 가진 개방주소지정하쉬표

5-2. 련습문제 5-1에서의 하쉬표들을 재하쉬한 결과를 표시하시오.

5-3. 선형탐색법, 2차탐색법, 2중하쉬법을 리용하여 삽입들의 우연서렬에 발생하는 충돌수를 계산하는 프로그람을 작성하시오.

5-4. 사슬주소하쉬표에서 삭제들이 많이 실행되면 그 표는 상당히 비게 될수 있는데 이것은 기억공간을 랑비하게 한다. 이 경우에 절반크기까지의 표에 대하여 재하쉬할수 있다. 표크기의 2배만한 세포들이 있을 때 더 큰 표를 재하쉬

한다고 하자. 더 작은 표를 재하쉬하기전에 표가 얼마나 비여 있겠는가.

5-5. 2차탐색법에서 isEmpty가 서술되지 않았다. currentSize==0표현을 되돌리는것으로써 그것을 실현할수 있는가?

5-6. 2차탐색하쉬표에서 findPos가 가리키는 위치에 새로운 항목을 삽입하는것대신에 탐색경로에서 첫번째로 비여 있는 세포에 삽입한다고 하자(따라서 기본적으로 기억공간을 유지하는 삭제표시된 세포를 개선하는것이 가능하다).

ㄱ. 이 결과를 리용하여 삽입알고리듬을 수정하시오. 추가적인 변수를 가지고 그것과 충돌하는 첫번째로 빈 세포의 위치를 유지하는 findPos에 의해 이것을 수행하시오.

ㄴ. 초기알고리듬보다 더 빠른 수정된 알고리듬에 관한 구체적인 내용을 설명하시오. 그 알고리듬은 속도가 더 떠질수 있는가.

5-7. 프로그람 5-3의 하쉬함수는 순환에서 key.length()를 반복적으로 호출한다. 순환에 들어 가기 직전에 이것을 계산할 필요가 있는가.

5-8. 여러가지 충돌처리방법들의 우점과 결함은 무엇인가.

5-9. 크기 M과 N에 대하여 두개의 성긴다항식 P_1과 P_2를 곱하는 다음의 방법을 실현하는 프로그람을 작성하시오. 매개 다항식은 결수와 지수로 구성되는 객체들의 련결목록으로 표현된다(련습문제 3-12). 전체 MN연산들에 대하여 P_1에 있는 매개 항목을 P_2에 있는 하나의 항목에 곱한다. 한가지 방법은 이 항목들을 정렬하고 같은 항목들을 결합하는것이지만 이것은 MN개의 기록들을 정렬할것을 요구하는데 이것은 작은 규모의 기억기들에서 특별히 비용이 많이 들게 된다. 한가지 방법은 계산된 항목들을 합하고 그다음에 그 결과를 정렬한다.

ㄱ. 그 방법을 실현하는 프로그람을 작성하시오.

ㄴ. 만일 출력다항식이 대략 $O(M+N)$개의 항목들을 가지면 두 산법들의 실행시간은 얼마인가?

5-10. 맞춤법검사기는 입력파일을 읽어서 어떠한 직결사전에 없는 모든 단어들을 출력한다. 사전이 30,000개의 단어들을 포함하고 파일이 너무 커서 알고리듬이 그 입력파일에 대하여 하나의 단계만 수행할수 있다고 하자. 한가지 간단한 방법은 사전을 하쉬표에 읽어 들이고 읽어 진 매개 입력단어를 검사하는 것이다. 단어가 평균적으로 7개의 문자들이고 L+1개의 바이트에 길이가 L인 단어들을 보관할수 있고(이렇게 하면 기억공간손실이 그리 많지 않다.) 2차탐색하쉬표를 리용한다고 하면 얼마나 많은 기억공간이 필요한가?

5-11. 만일 기억기가 제한되여 있고 전체 사전이 하쉬표에 보관될수 없다고 하여도 거의 언제나 동작하는 효과적인 알고리듬이 여전히 있을수 있다. 0부터 *TableSize*-1까지 bool값을 가지는 배렬 table(false로 초기화된)을 선언한다. 어

떠한 단어를 읽을 때 table[hash(word)] = true로 설정한다. 다음의 경우들에 대하여 어느것이 참으로 되는가?

ㄱ. 어떤 단어가 false값을 가진 위치에 하쉬되면 그 단어는 사전에 없다.

ㄴ. 어떤 단어가 true값을 가진 위치에 하쉬하면 그때 그 단어는 사전에 있다.

TableSize =300,007으로 선택한다.

ㄷ. 이것은 얼마나 많은 기억기를 요구하는가?

ㄹ. 이 알고리듬에서 오유가 나타날 확률은 얼마인가?

ㅁ. 일반적인 문서에서 500개의 단어들을 가지는 페지에 대하여 실지로 틀린 맞춤법은 약 3개이다. 이 알고리듬을 리용할수 있는가?

***5-12.** 기억기를 소비하여 하쉬표초기화를 무시하는 처리를 서술하시오.

5-13. 긴 입력문자렬 $A_1A_2 \cdots A_N$에서 문자렬 $P_1P_2 \cdots P_k$가 처음으로 발생하는 위치를 탐색하려고 한다고 하자. 이 문제는 견본문자렬을 하쉬하고 하쉬값 Hp를 얻으며 이 값을 $A_1A_2 \cdots A_k$, $A_2A_3 \cdots A_{k+1}$, $A_3A_4 \cdots A_{k+2}$, $\cdots$, $A_{N-k+1}A_{N-k+2} \cdots A_N$로 이루어 진 하쉬값과 비교하여 풀수 있다. 하쉬값들이 일치하면 그것을 확인하기 위하여 그 문자렬들을 한문자씩 비교한다. 문자렬들이 실제로 일치하면 그 위치(A에 있는)를 되돌리고 일치하지 않으면 처리를 계속해 나간다.

*ㄱ. 만일 $A_iA_{i+1} \cdots A_{i+k-1}$의 하쉬값을 알면 $A_{i+1}A_{i+2} \cdots A_{i+k}$의 하쉬값을 상수시간에 계산할수 있다는것을 증명하시오.

*ㄴ. 그 실행시간은 $O(k+N)$과 거짓일치들을 밝히는데 소비된 시간을 더한 것이라는것을 증명하시오.

*ㄷ. 예견되는 거짓일치들의 수가 극히 적다는것을 증명하시오.

*ㄹ. 이 알고리듬을 실현하는 프로그람을 작성하시오.

**ㅁ. 최악의 경우 $O(k+N)$시간에 실행되는 알고리듬을 작성하시오.

**ㅂ. 평균경우 $O(N/k)$시간에 실행되는 알고리듬을 작성하시오.

5-14. BASIC프로그람은 올리순서로 번호가 붙은 여러개의 지령들로 되여 있다. 그 조종은 goto나 gosub에 지령번호를 리용하여 분기된다. 적당한 BASIC프로그람을 읽어 들이고 처음에 번호 F에서 시작해서 매개 지령이 앞단계의 지령보다 D만큼 더 큰 번호를 가지도록 지령들에 다시 번호를 붙이는 프로그람을 작성하시오. 여기서 N개 지령들의 웃한계를 취해도 되지만 그 입력에서 지령수는 32bit옹근수만큼 클수 있다. 그 프로그람은 선형시간에 실행되여야 한다.

5-15. ㄱ. 이 장의 끝에서 서술한 알고리듬을 리용하여 단어맞추기프로그람을 작성하시오.

ㄴ. 매개 단어 W를 추가할 때 모든 W의 앞배치표현들을 정렬하여 속도를 크게 증가할수 있다. (만일 어떠한 W의 앞배치표현들이 사전에 없는

단어이면 그것은 실지 단어로써 보관된다.) 이것이 하쉬표의 크기를 많이 증가시키는것처럼 보이지만 많은 단어들이 같은 앞배치표현들을 가지기때문에 그렇게 되지 않는다. 조사가 각이한 방향에서 수행될 때 탐색된 단어가 앞배치처럼 하쉬표가 균일하지 않으면 그 방향에서의 조사는 빨리 끝낼수 없다. 이것을 리용하여 단어맞추기문제를 푸는 개선된 프로그람을 작성하시오.

ㄷ. 완전히 정확한 하쉬표ADT를 제공하려고 한다면 실례로《우수한》하쉬함수를 옳바로 계산해 놓았다면 처음부터《우수한》하쉬함수를 계산할 필요가 없다는것에 주의하여 ㄴ부분에 있는 프로그람의 속도를 개선할 수 있다. 이미전의 계산들의 우점을 가질수 있도록 하쉬함수를 수정하시오.

ㄹ. 제2장에서는 2진탐색을 보았다. 앞붙이를 리용하는 방법을 2진탐색알고리듬에 실현하시오. 그 수정은 간단한다. 어느 알고리듬이 더 빠른가?

5-16. 적당한 가정하에서 2차묶음을 가진 하쉬표에 대한 삽입의 기대값은 1/(1-λ)-λ-ln(1-λ)로 주어 진다. 그러나 이 식은 2차탐색법에 대하여 정확하지 않다. 그러나 그렇게 된다면 다음의것을 결정하시오.

ㄱ. 비성공적인 탐색의 기대값

ㄴ. 성공적인 탐색의 기대값

5-17. insert(삽입)와 lookup(찾아보기)연산들을 제공하는 일반적인 dictionary를 실현하시오. 이 실현은 쌍하쉬표(열쇠, 정의)를 보관한다. 사용자는 어떠한 열쇠(key)를 줌으로써 그에 대한 정의(difinition)를 찾아 보게 된다. 프로그람 5-14는 *dictionary* 명세서를 제공한다(일부 세부항목들은 없다.).

```
template <class HashedObj, class Object>
class Pair
{
    HashedObj key;
    Object      def;
        // Appropriate Constructors, etc.
};

template <class HashedObj, class Object>
class Dictionary
{
      public:
      Dictionary( );
      void insert( const HashedObj & key, const Object & definition );
      const Object & lookup( const HashedObj & key ) const;
      bool    isEmpty( ) const;
      void makeEmpty( );
private:
        HashTable<Pair<HashedObj ,Object> > items;
   ,
```

프로그람 5-14. 련습문제 5-17에 대한 사전구조

5-18. 하쉬표를 리용하여 맞춤법검사프로그람을 실현하시오. 사전은 다음의 두개의 원천 즉 존재하는 대사전과 개인용사전이 들어 있는 보조파일에서 나온것이라고 가정하시오. 모든 틀린 맞춤법단어들과 그것이 발생하는 행번호들을 출구하시오. 또한 매 틀린 맞춤법단어들에 대하여서는 다음의 규칙가운데서 임의의것을 적용해서 얻을수 있는 사전의 임의의 단어들을 게시하시오.

ㄱ. 한개의 기호를 첨부하시오.

ㄴ. 한개의 기호를 제거하시오.

ㄷ. 이웃 기호끼리 바꾸시오.

5-19. 하쉬자료구조내에 열쇠단어 10111101, 00000010. 10011011, 10111110, 01111111, 01010001, 10010110, 00001011, 11001111, 10011110, 11011011, 00101011, 01100001, 11110000, 0110111을 삽입하는 결과를 보여 주시오. 여기서 하쉬자료구조는 초기에는 비여 있으며 확장할수 있는것으로서 M=4이다.

5-20. 확장할수 있는 하쉬를 실행하도록 프로그람을 작성하시오. 표가 주기억기에 넣을수 있을만큼 충분히 작다면 사슬주소법과 개방주소지정하쉬표를 가지고 그의 성능을 어떻게 비교하겠는가?

참고문헌

하쉬가 명백히 단순함에도 불구하고 대부분의 분석은 아주 어렵고 여전히 많은 미지의 문제들이 있다. 또한 흥미 있는 리론적인 문제들도 많이 있다. 이 론점은 일반적으로 하쉬법에서 최악의 경우가 발생하지 않도록 하는것이다.

하쉬에 관한 초기론문은 [16]에 있다. 선형탐색법을 가진 하쉬법분석을 포함하여 문제에 대한 풍부한 내용은 [11]에서 얻을수 있다. [14]에서는 문제에 대하여 충분하게 개괄하고 있다. [15]는 하쉬함수를 선택하기 위한 방법을 포함하고 있다. 이 장에서 서술된 모든 방법들에 대한 정확한 해석과 모의결과들은 [8]에서 찾을수 있다.

2중하쉬법에 대한 분석은 [9]와 [13]에서 찾을수 있다. 또 다른 충돌처리전략은 [17]에서 서술된것처럼 무리로 존재하지 않는 균일한 평등하쉬가 성과적인 탐색의 비용면에서 최량이라는것을 보여 준다.

만약 입력열쇠들을 미리 알고 있을 때 충돌을 허용하지 않는 완전한 하쉬함수들은 [2]와 [7]에 서술되였다. 또한 복잡한 하쉬표들은 [3]과 [4]에 제기되였다(이 도식들에서 가장 최악의 경우는 개별적인 입력에 관계되지 않지만 알고리듬에 의하여 선택된 우연수들에는 관계된다.).

확장하쉬법은 [5]에 있으며 [6]과 [18]에서는 그에 대한 분석을 제기하였다.

련습문제 5-13 ㄱ~ㄹ는 [10]에서 참고하였다. ㅁ부분은 [12]로부터, ㅂ부분은 [1]로

부터 참고하였다.

1. R. S. Boyer and J. S. Moore, "A Fast String Searching Algorithm," *Communications of the ACM,* 20 (1977), 762-772.
2. J. L. Carter and M. N. Wegman, "Universal Classes of Hash Functions," *Journal of Computer and System Sciences,* 18 (1979), 143-154.
3. M. Dietzfelbinger, A. R. Karlin, K. Mclhorn, F. Meyer and der Heide, H. Rohnert, and R. E. Tarjan, "Dynamic Perfect Hashing: Upper and Lower Bounds," *SIAM Journal on Computing,* 23 (1994), 738-761.
4. R.J. Enbody and H. C. Du, "Dynamic Hashing Schemes," *Computing Surveys,* 20 (1988), 85-113.
5. R. Fagin, J. Nievergelt, N. Pippenger, and H, R. Strong, "Extendible Hashing—A Fast Access Method for Dynamic Files," ACM *Transactions on Database Systems, 4* (1979), 315-344.
6. P. Flajolet, "On the Performance Evaluation of Extendible Hashing and Trie Searching," *Acta Informatica* 20 (1983), 345-369.
7. M. L. Eredman, J. Komlos, and E. Szemeredi, "Storing a Sparse Table with O(1) Worst Case Access Time," *Journal of the ACM,* 31 (1984), 538-544.
8. G. H. Gonnet and R. Baeza-Yates, *Handbook of Algorithms and Data Structures,* 2nd ed., Addison-Wesley, Reading, Mass., 1991.
9. L. J. Guibas and E. Szemeredi, "The Analysis of Double Hashing," *journal of Computer and System Sciences,* 16 (1978), 226-274.
10. R. M. Karp and M. O. Rabin, "Efficient Randomized Pattern-Matching Algorithms," *Aiken Computer Laboratory Report TR-31-81,* Harvard University, Cambridge, Mass., 1981.
11. D. E. Knuth, *The Art of Computer Programming, Vol .3: Sorting and Searching,* 2d ed., Addison-Wesley, Reading, Mass., 1998.
12. D. E. Knuth, J. H. Morris, and V. R. Pratt, "Fast Pattern Marching in Strings," *SIAM Journal on Computing,* 6 (1977), 323-350.
13. G. Lueker and M. Molodowitch, "More Analysis of Double Hashing," *Proceedings of the Twentieth ACM Symposium on Theory of Computing* (1988), 354-359.
14. W. D. Maurer and T. G. Lewis, "Hash Table Methods," *Computing Surveys*, 7(1975), 5-20
15. B. J. Mckenzie, R. Harries, and T.Bell, "Selecting a Hashing Algorithm," *Software-Practice and Experience*, 20(19990), 209-224
16. W. W. Peterson, "*Addressing* for Random Access Storahe," IBM *Journal of Research and Development*, 1(1957), 130-146
17. J. S. Vitter, "Implementations for Coalesced Hashing," Communications *of the* ACM, 25(1982), 911-926
18. A. C. Yao, "A Note in The Analysis of Extendible Hashing," *Information Processing Letters*, 11(1980), 84-86
19. A. C. Yao, 'Uniform Hashing Is Optimal, "*Journal of the* ACM, 32(1985), 687-693.

제6장. 우선권대기렬(더미)

일반적으로 인쇄기에 전송되는 일감들은 **대기렬**에 배치되는데 이것이 언제나 제일 좋은것으로 되는것은 아니다. 례를 들어 어떤 일감이 특별히 중요하여 인쇄기에서 그 일감을 먼저 실행시키고 싶을 때가 있다. 반대로 만약 인쇄기를 리용할 1페지로 되여 있는 여러개의 일감들과 하나의 100페지짜리 일감이 있다면 큰 일감은 제일 마지막일감이 아니라고 하여도 마지막에 처리하는것이 합리적이다(그런데 이렇게 하는것은 때때로 성가실수 있기때문에 대부분의 체계들은 이렇게 하지 않는다.).

이와 류사하게 다중사용자환경에서 조작체계일정작성기는 여러개의 처리들가운데서 어느것을 수행해야 하는가를 결정해야 한다. 일반적으로 처리는 오직 고정된 시간주기내에서만 실행되도록 허용한다. 하나의 알고리듬은 대기렬을 리용한다. 일감들은 초기에 대기렬끝에 배치한다. 일정작성기는 반복적으로 대기렬에 있는 첫번째 일감을 취하여 그것이 끝나거나 그의 시간한계에 도달할 때까지 실행하며 일감이 끝나지 않았으면 그 일감을 대기렬끝에 배치한다. 이 방법은 일반적으로 적당치 않다. 왜냐하면 매우 작은 일감들이 기다림때문에 실행시간이 오래 걸리는것처럼 보이기때문이다. 일반적으로 작은 일감은 될수록 빨리 끝내는것이 중요하다. 따라서 이런 일감들은 이미 실행되고 있는 일감들보다 우선권을 가져야 한다. 더 나아가서 작지는 않지만 중요한 일부 일감들도 역시 우선권을 가져야 한다.

이런 응용프로그람들에서는 **우선권대기렬**(*priority queue*)이라고 하는 특수한 종류의 대기렬을 요구한다.

- 이 장에서 다음과 같은 문제들을 학습한다.
- 우선권대기렬 ADT의 효과적인 실현
- 우선권대기렬의 리용
- 우선권대기렬의 개선된 실현

이 자료구조는 콤퓨터과학에서 가장 품위 있는것이라고 볼수 있다.

제1절. 모형

우선권대기렬은 적어도 다음과 같은 두개의 연산 즉 명백한 일을 수행하는 insert와 우선권대기렬에서 최소인 요소들을 찾아 되돌려 보내며 삭제시키는 deleteMin을 가져야 한다. insert연산은 enqueue연산과 동등하고 deleteMin연산은 대기렬의 dequeue연산과 동등한것이다.

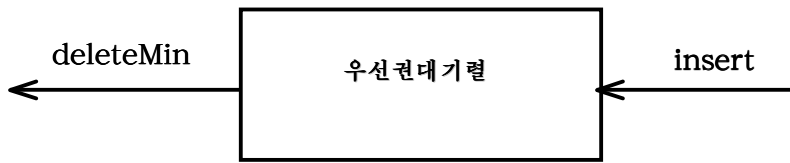



그림 6-1. 우선권대기렬의 기본모형

대부분의 자료구조에서처럼 때때로 다른 조작들을 추가할수 있지만 이것들은 우선권대기렬의 확장으로 되지 그림 6-1에서 묘사된 기본모형에 속하는 부분은 아니다. 우선권대기렬은 조작체계외에도 많은 응용프로그람들에서 리용된다. 제7장에서는 우선권대기렬들이 외부정렬에 어떻게 리용되는가를 고찰한다.

또한 우선권대기렬들은 **탐욕알고리듬**(*greedy algorithm*)의 실행에서도 중요하게 리용되는데 탐욕알고리듬은 최소값을 반복하여 찾는 연산을 수행한다. 제9장과 제10장에 이 실례들을 주었다. 이 장에서는 불련속적인 **사건모의**에서 우선권대기렬의 리용을 고찰한다.

제2절. 우선권대기렬의 간단한 실현

우선권대기렬을 여러가지 방법으로 실현할수 있다. 첫번째 방법은 간단한 련결목록을 리용하는것이다. 여기서는 $O(1)$시간에 대기렬의 앞위치에 요소들을 **삽입**하고 $O(N)$시간에 최소값을 삭제하기 위하여 목록을 순회하는 간단한 련결목록을 리용한다. 두번째 방법은 그 목록이 언제나 정렬되여 있을것을 요구하는것으로서 이것은 $O(N)$시간이 걸리는 삽입연산과 $O(1)$로써 상수적인 deleteMin연산을 수행한다. 첫번째 방법은 이 두가지 방법가운데서 더 좋은 방법으로 되고 있는데 그것은 실제로 삭제보다 삽입이 더많이 진행되기때문이다.

우선권대기렬실현의 또 하나의 방법은 2진탐색나무를 리용하는것이다. 이것은 우의 두 연산들에 대해 $O(\log N)$평균실행시간을 준다. 이것은 삽입은 우연적이고 삭제는 우연적이지 않음에도 불구하고 옳은것으로 된다. 삭제하려는 요소는 언제나 유일하게 최소값뿐이라는것을 상기하면 우선권대기렬에서의 련속적인 삭제는 왼쪽 부분나무에 있는 매듭들을 반복삭제하여 오른쪽 부분나무를 무겁게 만듦으로써 그 나무의 균형을 파괴시킨다. 그러나 오른쪽 부분나무는 우연적이다. 최악의 경우에 deleteMin이 왼쪽 부분나무를 삭제하면 오른쪽 부분나무는 많아서 삭제회수의 두배만한 요소들을 가지게 된다. 이것은 기대되는 **깊이**에 작은 상수만큼 추가한다. 평형나무를 리용하면 그 한계가 최악의 경우의 한계로 되는데 이것은 우선권대기렬을 좋지 못한 삽입렬로부터 보호한다.

탐색나무를 리용하는것은 필요 없는 연산들을 많이 주기때문에 그리 좋지 못하다. 여기에서 고찰하게 될 기본자료구조는 련결을 요구하지 않으며 최악의 경우에 $O(\log N)$인 두개의 연산을 지원한다. 삽입은 실제로 평균적인 상수시간을 가지게 될것이며 이 실현은 삭제가 요구되지 않는다면 선형시간에 N개 항목들을 가지는 우선권대기렬을 만들수 있다. 이제 효과적인 병합을 지원하는 우선권대기렬을 실현하는 방법을 설명하게 된다. 이 추가적인 연산으로 하여 문제는 약간 복잡한것처럼 보이며 외견상 련결구조를 리용한다.

제3절. 2진더미

여기에서 실현하게 되는 자료구조를 **2진더미**(*binary heap*)라고 한다. 그의 실현은 우선권대기렬의 실현과 같기때문에 우선권대기렬의 표현에서 단어 heap가 수식어구없이 리용될 때 일반적으로 그 자료구조의 실현에 대한 참조로 가정한다. 이 절에서는 2진더미를 그냥 **더미**(*heap*)라고 한다. 2진탐색나무에서처럼 더미들은 두가지 속성 즉 구조적속성과 **더미순서속성**을 가진다. AVL나무에서처럼 더미에 대한 연산은 그 속성들중의 하나를 무효로 할수 있다. 그러므로 더미연산은 모든 더미속성들이 차례로 처리될 때까지 끝나지 말아야 한다. 이것은 처리를 단순하게 한다.

1. 구조속성

더미는 2진나무이며 그 2진나무는 가장 깊은 준위를 제외하고는 완전히 꽉 채워 져 있다. 여기서 가장 깊은 준위는 왼쪽에서부터 오른쪽으로 채워 진다. 이와 같은 나무를 **완전2진나무**(*complete binary tree*)라고 한다. 그 실례를 그림 6-2에 보여 주었다.

높이가 h인 완전2진나무는 2^h과 $2^{h+1}-1$사이의 매듭을 가진다. 이것은 완전2진나무의 높이가 $\lfloor \log N \rfloor$, 명백히는 $O(\log N)$라는것을 암시한다.

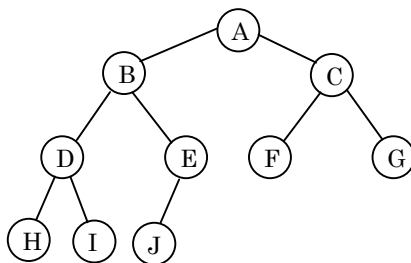



그림 6-2. 완전 2 진나무

중요한 문제는 완전2진나무가 너무 규칙적이기때문에 그것은 배렬로 표현될수 있으며 어떤 련결도 필요 없다는것이다. 그림 6-3에 있는 배렬은 그림 6-2에 있는 더미에 대응한다.

그림 6-3. 완전2진나무의 배렬실현

배렬위치 *i*에 있는 임의의 원소에 대하여 왼쪽 자식은 위치 2*i*에 있으며 오른쪽 자식은 위치 2*i*+1에 있다. 부모는 위치 *i*/2에 있다. 그러므로 련결을 요구하지 않을뿐만아니라 그 나무를 순회하는 연산은 극히 단순하며 대부분의 콤퓨터들에서 매우 빠르다.

더미실현에 대한 문제는 최대더미의 크기를 결정할것을 요구하지만 이것은 문제로 되지 않는다(만일 더 큰 더미가 요구되면 크기를 수정할수 있다.). 그림 6-3에서 더미크기는 13개의 요소로 제한한다. 배렬은 위치 0을 가진다. 더미자료구조는 배렬과 현재의 더미크기를 표현하는 옹근수로 이루어 진다. 프로그람 6-1에서는 우선권대기렬의 대면부를 보여 주었다.

이 장의 전반에서는 더미를 나무처럼 표시한다. 이것은 더미에 대한 실제적인 실현이 간단한 배렬들을 리용하는것과 같다는것을 의미한다.

```
template <class Comparable>
class BinaryHeap
{
    public:
        explicit BinaryHeap(    int capacity = 100 );
        bool isEmpty( ) const;
        bool isFull( ) const;
        const Comparable & findMin( ) const;
        void insert( const Comparable & x );
        void deleteMin( );
        void deleteMin( Comparable & mintem );
        void makeEmpty( );
   private:
        int                    currentSize;      // Number of elements in heap
        vector<Comparable> array;             // The heap array

        void buildHeap( );
        void percolateDown( int hole );
};
```

프로그람 6-1. 우선권대기렬의 클라스대면부

2. 더미의 순서속성

연산이 빨리 실행되게 하는 속성이 바로 **더미순서**(*heap-order*)속성이다. 최소값을 빨리 찾기 위해서는 가장 작은 요소가 뿌리에 있어야 한다. 만약 임의의 부분나무가 또 더미여야 한다고 하면 임의의 매듭은 그의 모든 자손들보다 더 작아야 한다.

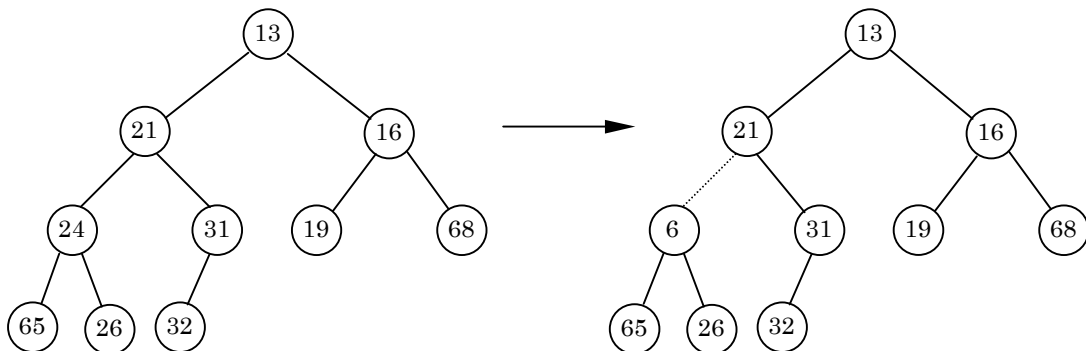



그림 6-4. 두개의 완전나무(왼쪽 나무만이 더미이다.)

이 원리를 적용하여 더미순서속성을 보장한다. 더미에서 그 뿌리(부모가 없는)를 제외하고 모든 매듭 *X*에 대하여 *X*의 부모에 있는 열쇠는 *X*에 있는 열쇠보다 더 작거나 같아야 한다.[19] 그림 6-5에서 왼쪽에 있는 나무는 더미이지만 오른쪽우에 있는 나무는 더미가 아니다(점선은 더미순서의 위반을 보여 준다.).

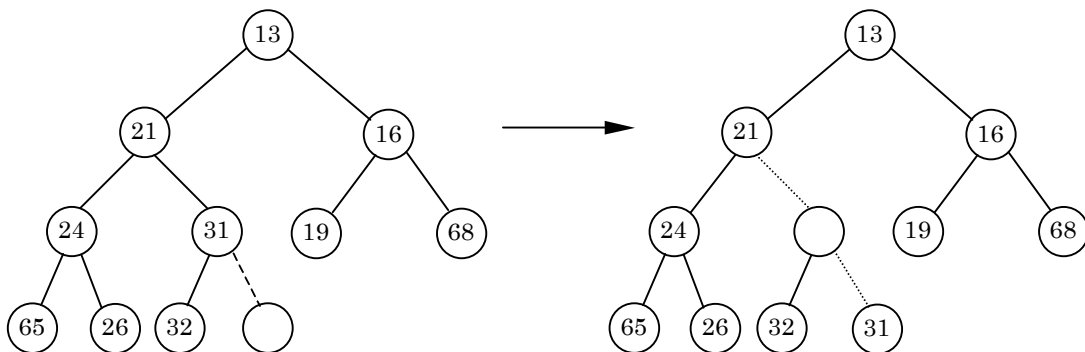



그림 6-5. 14에 대한 삽입시도: 구멍만들기 및 구멍을 우로 올려 보내기

---

[19] 이와 류사하게 우리는 최대더미를 정의할수 있다. 이 최대더미는 더미순서속성을 변화시키는것으로서 가장 큰 요소를 효과적으로 찾거나 제거할수 있게 한다. 따라서 우선권대기렬을 최대값 혹은 최소값을 찾는데 리용할수 있지만 이것은 미리 결정하여야 할 필요가 있다.

더미순서속성에 의하여 가장 작은 요소는 항상 뿌리에서 찾게 된다. 그러므로 상수시간에 여분의 연산인 findMin을 얻는다.

3. 더미의 기본연산

두개의 기본적인 더미연산들을 수행하는것은 개념적으로나 실천적으로 쉽다. 모든 처리는 더미순서속성을 보존하여야 한다.

Insert연산

요소 X를 더미에 삽입하기 위해서는 나무가 완전하지 않다는데로부터 삽입할 위치에 구멍을 만든다. X가 더미순서를 위반함이 없이 그 구멍에 배치될수 있다면 거기에 배치한다. 만일 더미순서를 위반한다면 그 위치의 부모매듭에 있는 요소를 그 구멍으로 넣는다. 그러면 그 구멍은 뿌리를 향하여 우로 올라 온다.

이 과정을 X가 그 구멍에 배치될 때까지 계속한다. 그림 6-5는 14를 삽입하기 위하여 그때 리용할수 있는 더미위치에 구멍을 만드는것을 보여 준다. 그러나 그 위치에 14를 삽입하는것은 더미순서속성에 위반된다. 따라서 31은 자기 위치의 아래에 재배치된다. 이 방법은 14에 대한 정확한 위치를 찾을 때까지 그림 6-6에서 계속된다. 이 일반적인 방법을 **우로려파**(*percolate up*)라고 한다. 즉 새로운 요소가 정확한 위치를 발견할 때까지 더미를 계속 우로 려파한다. 삽입은 프로그람 6-2에서 보여 준 코드에 의해 쉽게 실행된다.

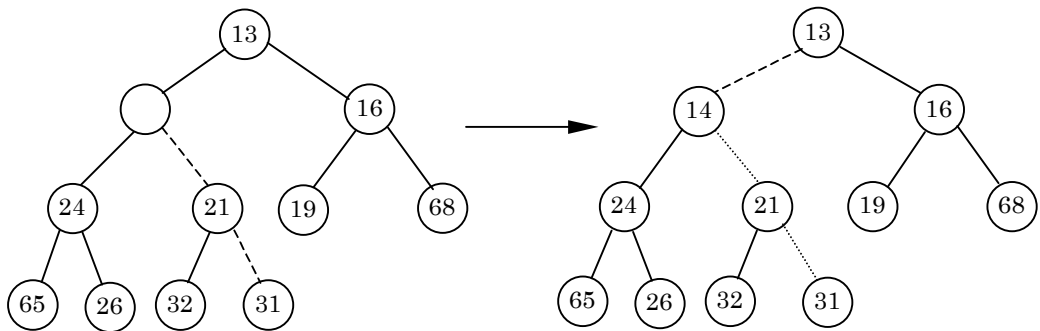



그림 6-6. 앞의 더미에 14를 삽입하기 위한 나머지 두 단계

여기에서는 정확한 순서가 설정될 때까지 교환을 반복 진행하는 방법으로 insert루틴내에서 려파를 실행하였지만 교환은 3개의 값주기명령들을 요구한다. 요소가 우로 d준위만큼 려파되였다면 교환을 위하여 수행된 값주기의 수는 $3d$일것이다. 이 방법은 $d+1$값주기를 리용한다.

```
/**
* Insert item x into the priority queue, maintaining heap order.
* Duplicates are allowed.
* Throw Overflow if container is full.
*/
template <class Comparable>
void BinaryHeap<Comparable>: :insert( const Comparable & x )
{
        if( isFull(   ) )
              throw Overflow( );

              // Percolate up
        int hole = ++currentSize;
        for( ; hole > 1 && x < array[ hole / 2 ]; hole /= 2 )
           array[ hole ] = array[ hole / 2 ];
        array[ hole] = x;
}
```

프로그람 6-2. 2진더미에서 삽입을 실현하기 위한 루틴

만일 삽입되는 요소가 가장 작은 새로운 값이면 그 값은 멀리 **꼭대기**(*top*)까지 넣어질것이다. 어떤 시점에서 hole은 1로 되여 순환고리로부터 탈퇴할것을 요구한다. 탈퇴는 명백한 검사로 수행할수 있다. 즉 순환끝을 만들기 위하여 매우 작은 값을 순서대로 위치 0에 놓을수 있다. 이 값은 더미에 있는 어떤 요소보다도 더 작아야 한다. 그것을 **감시매듭**(*sentinel*)이라고 한다. 이 방법은 련결목록안에 있는 선두매듭들의 리용과 류사하다. 정보에 대한 감시매듭을 추가함으로써 순환을 반복할 때마다 한번씩 실행되는 검사를 피할수 있으며 따라서 시간을 좀 더 절약할수 있다. 여기에서는 더미의 실현에 감시매듭을 리용하는것을 결정하지 않는다. 만약 삽입되는 요소가 새로운 최소값이고 멀리 뿌리까지 려과된다면 삽입하는데 걸리는 시간은 $O(\log N)$만큼 많이 든다.

평균적으로 려과는 빨리 끝난다. 즉 평균적으로 삽입을 수행하는데 2.607번의 비교가 요구된다. 그래서 평균적인 insert연산은 요소를 우로 1.607준위까지 이동시킨다.

deleteMin연산

deleteMin연산은 삽입과 류사한 방식으로 처리된다. 최소값을 찾는것은 쉬운데 정확한 처리는 그 최소값을 제거하는것이다. 최소값이 제거되면 뿌리에 구멍이 생긴다. 이제 더미가 하나의 더 작은 더미로 된다는데로부터 더미내의 마지막요소 X는 더미내의 어떤 곳으로 이동되여야 한다. 만약 X가 그 구멍내에 배치될수 있다면 처리는 원만히 수행되였다고 본다. 만일 X가 그 구멍내에 배치될수 없다면 구멍의 두 자식들중에서 더 작은것을 그 구멍에 넣으며 따라서 그 구멍은 아래로 한준위 내려 간다. 이 단계를 X가 그 구멍에 배치될수 있을 때까지 반복한다. 따라서 가장 작은 자식을 포함하는 뿌리로부터 경

로를 따라 *X*를 그의 정확한 위치에 배치한다.

그림 6-7에서 왼쪽 그림은 deleteMin연산을 수행하기전의 더미를 보여 준다. 13이 제거된후 이제는 31를 더미에 배치하여야 한다. 값 31은 그 구멍에 배치될수 없는데 그것은 더미순서를 위반하기때문이다. 그러므로 그 구멍을 아래로 한준위 내려 보내면서 구멍에 더 작은 자식 14를 배치한다(그림 6-8). 이것을 다시 반복하여 31이 19보다 더 크기때문에 그 구멍에 19를 배치하며 새 구멍을 한준위 더 깊게 만든다. 그러면 26이 그 구멍에 배치되며 31이 더 크기때문에 다시 한번 새 구멍을 맨 밑준위에 만든다. 마지막으로 31을 그 구멍에 배치할수 있다(그림 6-9). 이러한 일반적인 방법을 내리려과(*percolate down*)라고 한다. 이 루틴에서 교환을 쓰지 않기 위해 insert루틴과 같은 수법을 리용한다.

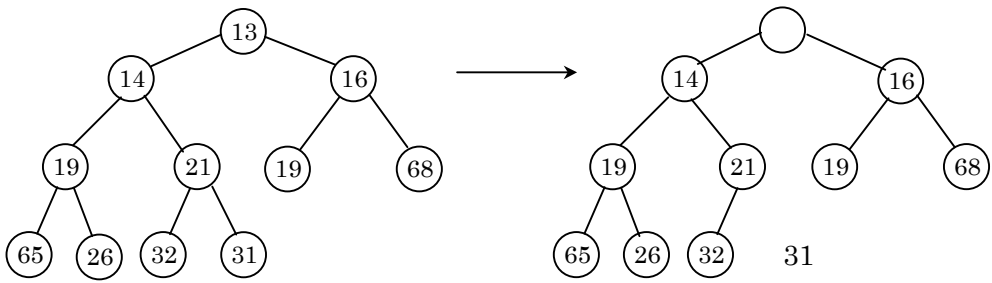



그림 6-7. 뿌리에 구멍만들기

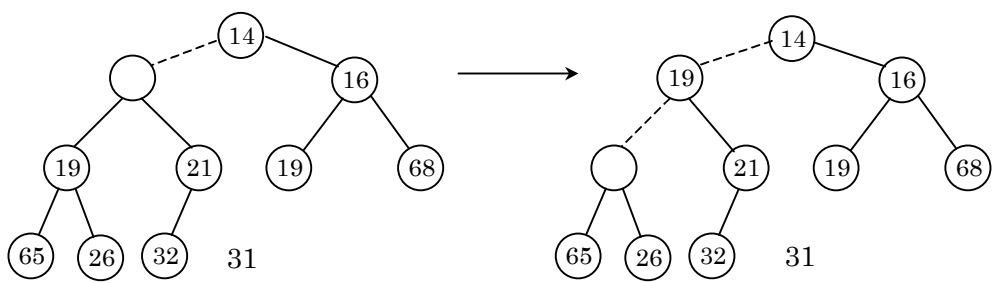



그림 6-8. deleteMin에서의 다음의 두 단계

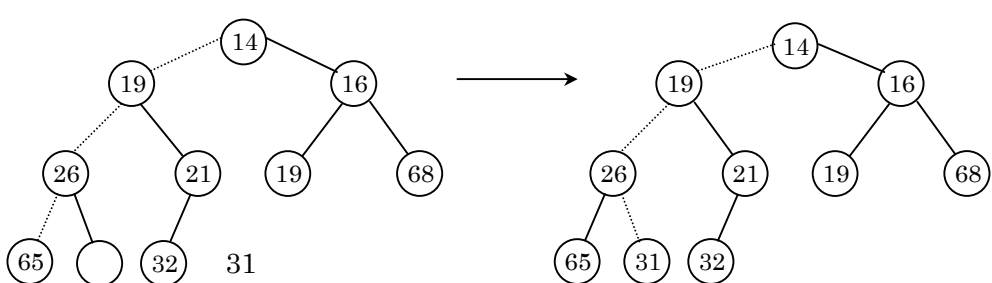



그림 6-9. deleteMin안에서의 마지막 두 단계

더미들에서 흔히 발생하는 실행오유는 더미에 짝수개의 원소들이 있을 때 즉 오직 하나의 자식만 가지는 매듭이 있을 때이다. 매개 매듭이 항상 두개의 자식을 다 가진다고 생각하지 말아야 한다. 따라서 보통 이에 대한 여분의 검사 또는 특별한 검사를 진행하여야 한다. 프로그람 6-3에서 보여 준 코드에서는 이러한 검사를 5행에서 수행한다. 한가지 아주 재치 있는 풀기방법은 사용자알고리듬이 늘 매개 매듭이 두개의 자식을 가졌다고 생각하는것이다. 이것은 감시매듭을 배치하여 처리한다. 이때 감시매듭은 더미의 임의의 값보다 더 큰것이며 더미에서 끝위치 다음에 놓인다. 그러나 더미크기가 짝수일 때는 내리려파의 시작위치에 놓는다. 사용자는 이것을 시도하기전에 매우 주의깊게 생각하여야 하며 만약 이 수법을 리용할 때에는 주해를 달아야 한다. 비록 이것은 오른쪽 자식의 존재에 대한 검사는 하지 않는다 하더라도 사용자는 매개 잎매듭에 대한 감시매듭을 요구하기때문에 더미의 바닥에 도달할 때까지 검사를 해야 한다.

```
    /**
    * Remove the smallest item from the priority queue
    * and place it in minItem. Throw Underflow if empty.
    */
    template <class Comparable>
    void BinaryHeap<Comparable>::deleteMin( Comparable & minItem )
    {
       if( isEmpty( ) )
          throw Underflow( );
       minItem = array[ 1 ];
       array[ 1 ] = array[ currentSize - - ];
       percolateDown( 1 );
    }

    /**
    * Internal method to percolate down in the heap.
    * hole is the index at which the percolate begins.
    */
    template <class Comparable>
    void BinaryHeap<Comparable>::percolateDown( int hole )
    {
/*1*/     int child;
/*2*/     Comparable tmp = array[ hole ];
/*3*/     for( ; hole * 2 <= currentSize; hole = child )
          {
/*4*/         child = hole    * 2;
/*5*/         if( child != currentSize && array[ child + 1 ] < array[ child ] )
/*6*/             child++;
/*7*/         if( array[ child ] < tmp )
/*8*/             array[ hole ] = array[ child ];
```

```
                        Else
/*9*/                         break;
                   }
/*10*/      array[ hole ] = tmp;
       }
```

프로그람 6-3. 2진더미에서 deleteMin을 수행하는 산법

이 연산에 대한 최악의 경우의 실행시간은 $O(\log N)$이다. 평균경우에 뿌리에 배치되는 요소는 거의 더미의 바닥까지 려과된다. 따라서 평균실행시간은 $O(logN)$으로 된다.

4. 더미의 기타연산

비록 최소값을 찾는 문제가 상수시간에 수행될수 있다고 해도 최소값을 찾도록 설계된 더미(min)는 최대값요소를 찾는데서 아무런 역할도 하지 못한다. 사실상 더미는 매우 작은 순서정보를 가지고 있으므로 더미전반을 선형적으로 조사하지 않고 어떤 개별적인 요소를 찾는 방법은 없다. 이것은 그림 6-10에 있는 큰 더미구조(요소들은 보여 주지 않았다.)를 보면 알수 있는데 여기에서 최대값요소에 대한 정보는 오직 잎매듭들중에 있다. 요소들의 절반정도가 잎매듭들이므로 이것은 실제적으로 리용할수 없는 정보이다. 이와 같은 원인으로 만약 요소들의 위치를 아는것이 중요하다면 하쉬표와 같은 어떤 다른 자료구조가 더미에 추가되여 리용되여야 한다(그 모형은 더미에서 취급하지 않는다.).

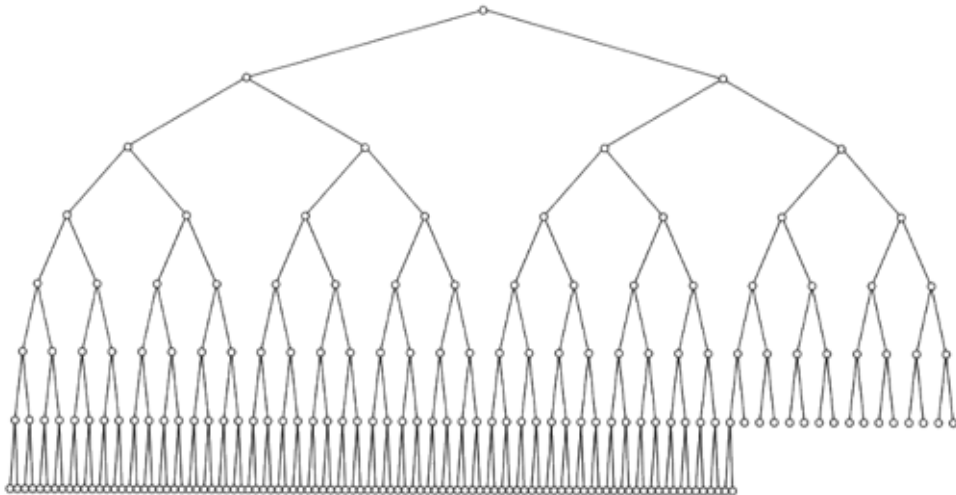



그림 6-10. 방대한 완전2진나무

만일 어떤 다른 방법에 의하여 모든 요소들의 위치를 알수 있다면 기타 여러 연산

들은 쓸데 없는것으로 된다. 다음의 첫 3개의 연산은 모두 최악의 경우 로그시간에 실행된다.

decreaseKey연산

decreaseKey(P, Δ)연산은 절대량 Δ에 의해 위치 p에 있는 항목의 값을 감소시킨다. 이것은 더미순서를 위반하기때문에 우로추출에 의해서 고정되여야 한다. 이 연산은 체계관리에 유용한데 그것은 체계프로그람들을 제일 높은 우선권을 가지고 실행할수 있게 한다.

increaseKey연산

IncreaseKey(P, Δ)연산은 절대량 Δ에 의하여 위치 p에 있는 항목의 값을 증가시킨다. 이것은 내리려과로 수행된다. 많은 일정작성기들은 과도적인 CPU시간을 소비하는 처리들에 대하여서는 우선권을 자동적으로 낮춘다.

remove연산

remove(p)연산은 더미에서 p위치에 있는 매듭을 제거한다. 이것은 처음에 decreasekey(p, ∞)를 수행하고 다음에 deleteMin()을 호출하여 수행한다. 처리가 사용자에 의해서 끝날 때(표준적으로 끝날대신) 그 매듭은 우선권대기렬로부터 제거되여야 한다.

buildHeap연산

buildHeap연산은 입력으로 N개 항목을 취하여 그것들을 빈 더미에 배치한다. 이것은 N개의 련속적인 insert연산들에 의하여 수행될수 있다. 매 삽입은 평균으로는 $O(1)$, 최악의 경우에는 $O(\log N)$을 취하므로 이 알고리듬의 총 실행시간은 평균적으로 $O(N)$이지만 최악의 경우에는 $O(N\log N)$이다. 이것은 특수한 명령이고 그 어떤 다른 연산들이 관계되지 않으며 또한 그 명령이 평균시간에 선형으로 실행될수 있다는데로부터 그 연산은 론리적으로 선형시간한계가 담보된다는것을 알수 있다.

일반적인 알고리듬은 N개의 항목을 구조속성을 보존하면서 그 나무에 어떤 순서로 배치하는것이다. 그때 percolateDown(i)가 매듭 i로부터 아래로 려과된다면 더미순서나무를 만들기 위하여 프로그람 6-4에 있는 알고리듬을 실행한다.[20]

그림 6-11에 있는 첫번째 나무는 순서화되지 않은 나무이다. 그림 6-11~6-14에 있는

---

[20] 이 코드는 더미순서속성을 위반할수 있는 그 어떤 일반적인 방법들이 없기때문에 가상코드로 된다. 이것을 수행하는 한가지 가능한 방법은 N개의 항목을 포함하고 있는 배렬을 넘기는것이며 이것들을 배렬내에 build더미복사를 하고 다음려과를 수행하는것이다.

나머지 7개의 나무들은 7번의 percolateDown연산들의 매개 실행결과를 보여 준다. 매 점선은 다음의 두가지 비교에 대응된다. 즉 하나는 더 작은 자식을 찾는것이고 다른 하나는 더 작은 자식을 그 매듭과 비교하는것이다. 20개의 비교에 대응하는 전체 알고리듬내에는 오직 10개의 점선만 있다는것을 고려해야 한다.

```
/ **
   *   Establish heap order property from an arbitrary
   *   arrangement of items. Runs in linear time.
   */
template <class Comparable>
Void BinaryHeap<Comparable>: :buildHeap( )
{
    for( int   i = currentSize / 2; i > 0; i-- )
        percolateDown( i );
}
```

프로그람 6-4. build Heap의 골격

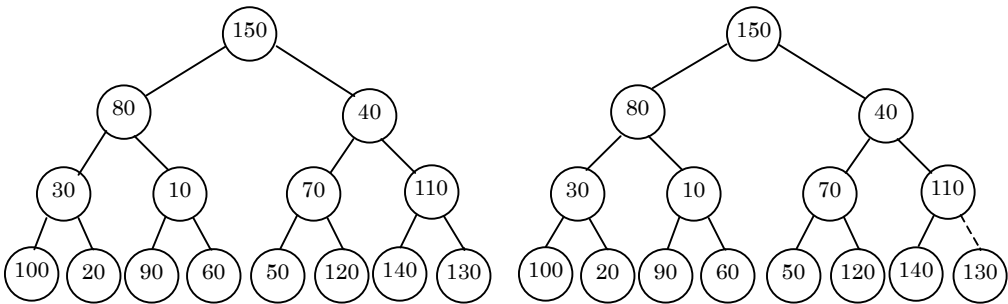



그림 6-11. 왼쪽: 초기더미; percolateDown(7)연산을 수행한후

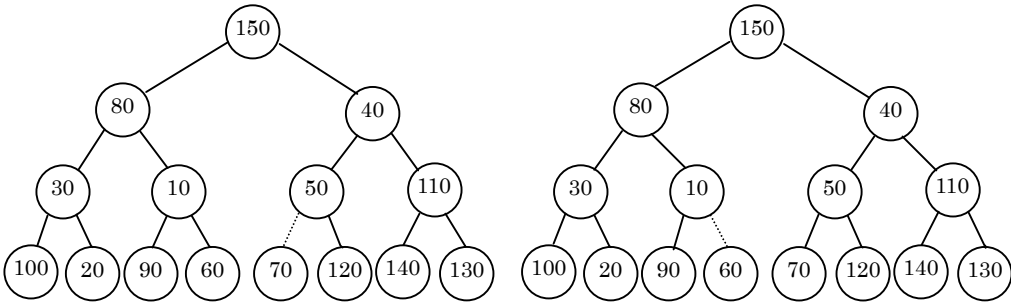



그림 6-12. 왼쪽: percolateDown(6)연산을 수행한후
오른쪽: percolateDown(5)연산을 수행한후

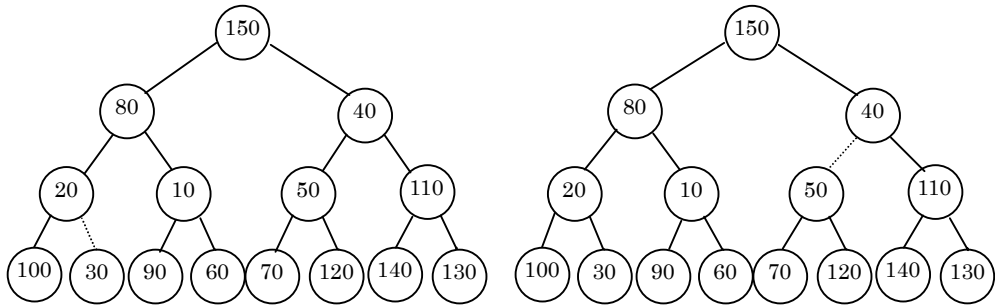



그림 6-13. 왼쪽: percolateDown(4)연산을 수행한후
오른쪽: percolateDown(3)연산을 수행한후

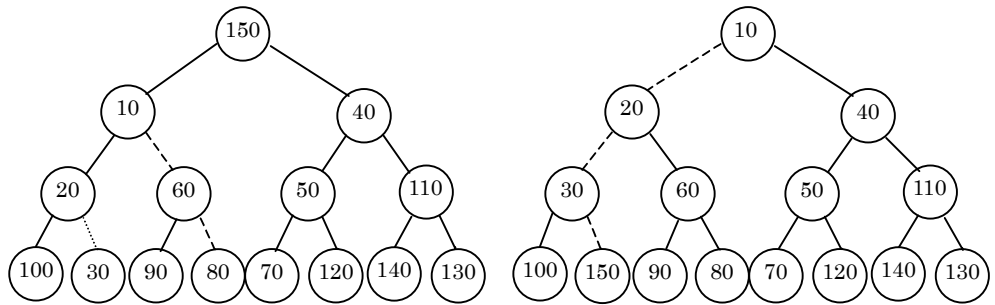



그림 6-14. 왼쪽: percolateDown(2)연산을 수행한후
오른쪽: percolateDown(1)연산을 수행한후

buildHeap의 실행시간을 제한하려면 점선의 개수를 제한하여야 한다. 이것은 더미내에 있는 모든 매듭의 높이의 합을 계산하여 수행될수 있는데 이 합은 점선들의 최대개수이다. 여기서 보여 주려는것은 이 합이 $O(N)$이라는것이다.

정리 6-1.

$2^{h+1}-1$개의 매듭을 포함하는 높이가 h인 완전2진나무에 대하여 매듭들의 높이의 합은 $2^{h+1}-1-(h+1)$이다.

증명:

이 나무는 높이 h에는 한개 매듭, 높이 h-1에는 두개 매듭, h-2에는 2^2매듭 그리고 일반적으로 높이 $h-i$에는 2^i개의 매듭들로 이루어 진다는것을 쉽게 알수 있다. 그때 모든 매듭들의 높이의 합은

$$S=\sum_{i=0}^{h} 2^i(h-i) = h+2(h-1)+4(h-2)+8(h-3)+16(h-4)+\cdots+2^{h-1}(1) \quad 6\text{-}1$$

여기에 2를 곱해서 다음의 같기식을 얻는다.

$$2S = 2h+4(h-1)+8(h-2)+16(h-3)+\cdots+2^h(1) \qquad 6\text{-}2$$

이 두 식을 덜어서 식 6-3을 얻는다. 그러면 거의 모든 항들이 소거된다는것을 알 수 있다. 실례로 $2h-2(h-1)=2$, $4(h-1)-4(h-2)=4$ 등
식 6-2에 있는 마지막항 2^h는 식 6-1에는 없다. 따라서 그것은 식 6-3에 남아 있다. 또한 식 6-1에 있는 첫번째 항 h는 식 6-2에 없으므로 $-h$도 식 6-3에 남아 있다. 따라서 식

$$S=-h+2+4+8+ \cdots +2^{h-1}+2^h =(2^{h+1} -1)-(h-1) \qquad 6\text{-}3$$

을 얻는다. 이렇게 하여 정리가 증명된다.

완전나무는 완전2진나무는 아니지만 여기서 얻은 결과는 완전나무에서 매듭들의 높이의 합에 대한 웃한계이다. 완전나무가 2^h과 2^{h+1}사이의 매듭들을 가진다는데로부터 이 정리는 이 합이 $O(N)$이라는것을 암시한다. 여기서 N은 매듭들의 개수이다.

비록 얻은 결과가 buildHeap연산이 선형적이라는것을 보여 준다하더라도 그 높이들의 합에 대한 한계는 그리 명백하지 않다. $N = 2^h$매듭들을 가지는 완전나무에서 그 한계는 대략적으로 $2N$이다. 그 높이들의 합은 $N-b(N)$로 되는데 여기서 $b(N)$은 N의 2진표현에 있는 1의 개수이다.

제4절. 우선권대기렬의 응용

우선권대기렬들이 조작체계설계에 어떻게 리용되는가 하는것은 이미 고찰하였다. 제9장에서는 우선권대기렬들이 여러가지 그라프알고리듬들을 효과적으로 실행하는데 어떻게 리용되는가에 대하여 보게 된다. 여기서는 2개의 문제풀이에 우선권대기렬을 리용하는 방법을 보게 된다.

1. 선택문제

여기서 론의할 첫번째 문제는 제1장에서 본 **선택문제**(*selection problem*)이다. 그 입력은 총체적으로 순서화될수 있는 N개의 요소들의 목록과 옹근수 k이다. 선택문제는 그 목록에서 k번째로 가장 큰 요소를 찾는것이다.

제1장에서는 두가지 알고리듬을 주었는데 어느것이나 다 매우 비능률적이다. 첫번째 알고리듬(이 알고리듬을 1A라고 하자.)은 요소들을 배렬에 읽어 들이고 그것들을 정렬

한 다음 적당한 요소를 되돌리는것이다. 단순한 정렬알고리듬을 리용하기때문에 그 실행시간은 $O(N^2)$이다. 이 문제에 대한 다른 알고리듬 1B는 k개의 요소들을 배렬에 읽어 들이고 그것들을 정렬한다. 이 요소들가운데서 제일 작은 요소는 k번째 위치에 놓이게 된다. 다음 나머지요소들을 하나씩 처리한다. 즉 매개 요소를 배렬에 있는 k번째 요소와 비교하여 만약 그것이 더 크다면 k번째 요소는 제거되고 새 요소 나머지 k-1개의 요소가운데서 정확한 위치에 배치된다. 알고리듬이 끝날 때 k번째 위치에 있는 요소가 얻으려는 결과이다. 그 실행시간은 $O(N \cdot k)$이다(왜 그렇게 되는가?).

$k=\lceil N/2 \rceil$라면 이 두 알고리듬은 $O(N^2)$이다. 임의의 어떤 k에 대하여 $(N\text{-}k+1)$번째로 가장 작은 요소를 찾기 위한 문제를 대칭적으로 풀수 있는데 $k=\lceil N/2 \rceil$은 이 알고리듬에 대한 가장 어려운 경우로 된다. 이것은 또한 가장 흥미 있는 경우로 되는데 여기서 이 k값을 **중위수**(*median*)라고 한다.

이제 두개의 알고리듬(6A와 6B)을 보자. 이 두 알고리듬은 모두 $k=\lceil N/2 \rceil$인 경우에 $O(N\log N)$으로 실행되며 이것은 독특한 개선으로 된다.

알고리듬 6A

간단히 k번째로 가장 작은 요소(smallest)를 찾는다고 하자. 이에 대한 알고리듬은 간단하다. 먼저 N개의 요소를 배렬에 읽어 들이고 그다음 이 배렬에 대하여 buildHeap 알고리듬을 적용한다. 마지막으로 deleteMin연산을 k번 수행한다. 더미로부터 선출된 마지막요소가 얻으려는 결과이다.

더미순서속성을 변경시켜 초기문제를 k번째로 가장 큰 요소를 찾는 문제로 만들수 있다. 그 알고리듬의 정확성은 명백하여야 한다. 최악의 경우 buildHeap를 리용하여 더미를 구축하는데 걸리는 시간은 $O(N)$이다. 그리고 매 deleteMin연산인 경우에는 $O(\log N)$이다. deleteMin연산이 k번 실행되므로 총 실행시간은 $O(N+k\log N)$으로 된다. 만약 $k=O(N+k\log N)$이라면 실행시간은 buildHeap연산에 의하여 $O(N)$으로 된다. 더 큰 값 k에 대하여 그 실행시간은 $O(k\log N)$이다.

$k=\lceil N/2 \rceil$이라면 실행시간은 $\Theta(N\log N)$이다.

만약 이 프로그람을 $k=N$에 대해 실행하고 요소들이 더미에서 선출될 때의 값들을 기록한다면 입력파일은 본질적으로 $O(N\log N)$시간에 정렬된다. 이 방법은 제7장에서 **더미정렬**이라고 하는 고속정렬알고리듬을 고찰할 때 자세히 설명하게 된다.

알고리듬 6B

두번째 알고리듬에서 본래의 문제로 되돌아 가서 k번째 가장 큰 요소(lagest)를 찾는다. 여기에서는 알고리듬 1B에서 고찰한 방법을 리용한다. 매 시점에서 k개의 가장 큰

요소들의 모임 S를 보존한다. 첫 k개의 요소들이 읽어 진 다음 새로운 요소가 읽어 질 때 그것은 k번째 가장 큰 요소와 비교되는데 이것을 S_k로 표시한다. S_k는 S안에 있는 제일 작은 요소라는데 주의하자. 새로운 요소가 더 크다면 그것을 S에 속하는 S_k로 치환한다. 그러면 S는 새로운 제일 작은 요소를 가지게 된다. 그 새로운 제일 작은 요소는 새로 첨부된 요소일수도 있고 그렇지 않을수도 있다. 입력렬의 끝에서 S에 속하는 제일 작은 요소가 곧 결과로 된다.

이것은 본질적으로 제1장에서 서술한것과 같은 알고리듬이다. 그러나 여기서는 S를 실행하기 위해 더미를 리용한다. 첫 k개의 요소들은 buildHeap에 대한 호출을 리용하여 총체적으로 $O(k)$시간에 더미에 배치된다. 나머지 매 요소들에 대한 처리시간은 $O(1)$이다. 이때 그 요소가 S에 넣어 지는가를 검사하기 위해서는 $O(\log k)$를 더한다. 그리고 S_k를 삭제하고 필요하다면 새 요소를 삽입한다. 따라서 총 시간은 $O(k+(N-k)\log k)=O(N\log k)$으로 된다. 이 알고리듬은 또한 중위수를 찾는데 $\Theta(N\log N)$의 시간한계를 준다.

제7장에서는 평균적으로 $O(N)$시간에 이 문제를 푸는 방법을 고찰한다. 제10장에서는 최악의 경우 $O(N)$시간에 이 문제를 풀기 위하여 비실용적이지만 효과적인 알고리듬을 고찰한다.

2. 사건모의

제3장 제4절 3에서 중요한 대중봉사문제를 서술하였다. 실례로 은행업무체계에 대하여 고찰해 보자. 주문자들은 은행에 도착해서 k명의 출납원들중의 한사람을 리용할수 있을 때까지 한줄로 대기한다. 주문자도착은 봉사시간(일단 출납원을 리용할수 있기만 하면 봉사받는 시간량)에 따르는 확률분포함수로 결정된다. 체계관리자는 주문자가 평균적으로 얼마나 오래 기다려야 하는가 또는 그 줄이 얼마나 긴가와 같은 통계에 흥미를 가진다.

이 인자들은 일정한 확률분포와 k값들을 가지고 정확히 계산될수 있다. 더우기 k를 더 크게 하면 분석은 상당히 더 어렵게 되며 따라서 은행업무를 모의하기 위해 콤퓨터를 리용하게 된다. 이런 방식으로 은행직원들은 적당하고 원활한 봉사를 보장하는데 몇명의 출납원들이 요구되는가를 결정할수 있다.

모의는 사건들의 처리로 이루어 진다. 여기에 두개의 사건들이 있다. 즉 주문자의 도착과 주문자의 떠나기이다. 주문자가 떠나면 출납원은 해제된다.

여기에서는 도착시간에 의해서 정렬된 매 주문자에 대한 도착시간과 봉사시간의 순서화된 쌍들로 이루어 진 하나의 입력흐름을 얻기 위하여 확률함수를 리용할수 있다. 이 체계에서는 하루의 정확한 시간을 알아야 할 필요가 없다. 그래서 일정한 단위를 리용하는데 이것은 **박자**(*tick*)로 처리할수 있다.

이 모의를 수행하는 한가지 방법은 모의시계를 0박자에서부터 시작하는것이다. 그래서 사건이 있는가를 검사하면서 그 시계를 한번에 한박자씩 전진시킨다. 만일 사건이 있다면 그 사건을 처리하고 통계량을 수집한다. 입력흐름에 어떤 주문자도 남아 있지 않고 모든 출납원들이 자유롭다면 모의는 끝난다.

이 모의방법에서는 그 실행시간이 주문자들이나 사건들의 수(매 주문자에 대하여 두개의 사건이 있다.)에 관계된다. 그리고 박자의 개수에도 관계되는데 이것은 실제적인 입력부분이 아니다. 이것이 왜 중요한가를 보기 위해서 시계단위들을 1/1000박자로 변화시키고 모든 입력시간들에 1000을 곱하자. 그 결과는 모의가 1000배나 더 오래 걸린다는 것을 나타낸다.

이러한 문제점을 피하는 방도는 매 단계에서 시계를 다음의 사건시간으로 전진시키는것이다. 이것은 개념적으로 리해하기 쉽다. 어떤 시점에서 발생할수 있는 다음의 사건은

ㄱ. 입력파일에 다음의 주문자가 도착하였다든가

ㄴ. 주문자들중의 한명이 출납원으로부터 떠났다는것이다.

사건들이 발생하는 모든 시간은 쓸모 있기때문에 앞으로부터 제일 먼저 발생하는 사건을 찾아서 그 사건을 처리하여야 한다.

만약 사건이 어떤 출발이라면 처리는 출발하는 주문자들을 위한 통계를 집합하고 또 다른 주문자가 기다리고 있는가를 보기 위하여 기다림렬(대기렬)을 검사한다. 그러자면 주문자를 추가하고 필요한 임의의 통계를 진행한다. 그다음 주문자가 떠나는 시간을 계산하여 차후에 진행하여야 할 작업항목에 포함시킨다.

사건이 하나의 도착이면 체계는 리용할수 있는 출납원을 검사한다. 만약 누구도 없다면 그 도착을 기다림렬(대기렬)에 놓으며 또한 리용할수 있는 출납원이 있으면 주문자를 출납원에게 보내고 주문자의 출발시간을 계산하여 차후에 진행하여야 할 작업항목에 포함시킨다.

주문자들을 위한 기다림렬은 하나의 대기렬로 취급할수 있다. 은행업무체계는 앞에서 제일 가까운 사건을 찾아야 하므로 발생을 기다리는 출발모임을 우선권대기렬로 만든다. 따라서 다음에 발생하는 사건은 다음의 도착이 아니면 다음의 출발인데 둘다 쉽게 리용할수 있다.

비록 있을수 있는 시간소비는 있더라도 그 모의루틴들은 간단하게 작성할수 있다. 만일 C명의 주문자들과 k명의 출납원이 있다면(따라서 $2C$개의 사건이 가능한) 계산과 처리의 매개 사건은 $O(\log H)$에 실행되므로 모의의 실행시간은 $O(C\log(k+1))$로 될것이다.[21] 여기서 $H=k+1$은 더미의 크기이다.

---

[21] k=1 인 경우에 대한 혼돈을 피하기 위하여 $O(C\log k)$대신에 $O(C\log(k+1))$을 리용한다.

제5절. *d*-더미

2진더미들은 너무도 단순하기때문에 그것들은 거의 언제나 우선권대기렬을 요구할 때 리용된다. 2진더미에 대한 하나의 단순한 일반화는 ***d*-더미**(*d-heap*)이다. 이 *d* -더미는 모든 매듭이 *d*개의 자식을 가진다는것을 제외하고는 2진더미와 완전히 같다(따라서 2진더미는 2-더미라고 할수 있다.).

그림 6-15는 3-더미를 보여 준다. *d*-더미는 2진더미보다 깊이가 상당히 얕다는것을 고려하여야 한다. 그래서 insert에 대한 실행시간은 $O(\log_d N)$으로 개선된다. 그러나 *d*가 클 때 deleteMin연산은 비용이 더 드는데 그것은 설사 나무가 더 얕다고 해도 *d*개의 자식들에 대한 최소값을 찾아야 하며 표준알고리듬을 리용하여 그것을 찾을 때 *d*-1개의 비교가 진행되기때문이다. 이것은 deleteMin연산시간을 $O(d\log_d N)$으로 증가시킨다.

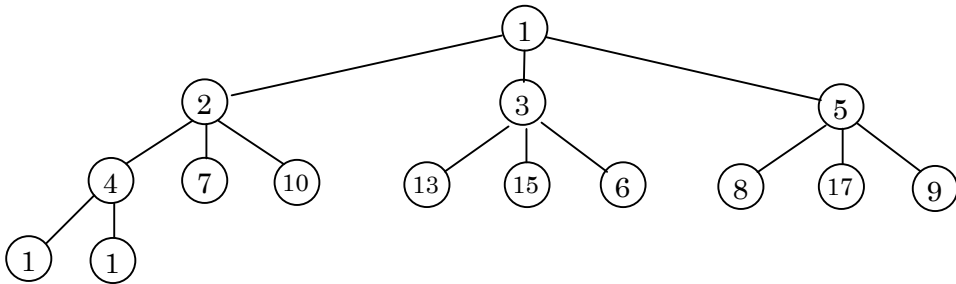



그림 6-15. 하나의 *d* -더미

*d*가 상수라면 실행시간은 물론 둘다 $O(logN)$이다. 비록 배렬을 여전히 리용할수 있다 하더라도 자식과 부모를 찾기 위하여 *d*로써 곱하기들과 나누기들을 진행하게 되는데 만일 *d*가 2의 제곱이 아닌이상 실행시간은 계속 증가한다. 그 리유는 비트밀기에 의해 나누기를 더이상 할수 없기때문이다. *d* -더미들은 리론적으로 흥미 있는데 그것은 대부분의 알고리듬에서 삽입의 수가 deleteMin연산의 개수보다 훨씬 더 많기때문이다(따라서 리론적인 속도개선이 가능하다.). 그것들은 또한 우선권대기렬이 너무 커서 주기억기에 완전히 넣을수 없을 때에도 흥미가 있다. 이 경우에 *d*-더미는 B나무와 같은 방식에서 훨씬 우월하다. 마지막으로 실천에서는 4-더미가 2진더미를 릉가한다는 견해도 제기되고 있다.

더미실현의 가장 큰 결함은 find연산의 실행불가능성외에 두개의 더미를 하나로 결합하는 연산이 어렵다는것이다. 이 특별한 연산을 **병합**(*merge*)이라고 한다. merge의 실행시간이 $O(logN)$이 되도록 더미를 실현하는 방법은 많지 못하다. 이제 여러가지 복잡성을 가진 세개의 자료구조를 설명하게 되는데 그것은 merge연산을 효률적으로 지원한다. 모든 복잡한 분석은 제11장에서 진행한다.

제6절. 왼쪽더미

병합을 효과적으로 지원하면서도 2진더미에서처럼 배렬만을 리용하는 자료구조(이것은 merge를 $O(N)$시간에 처리한다.)를 설계한다는것은 어렵다. 그 리유는 병합이 하나의 배렬을 또 다른 배렬에 복사할것을 요구하는것처럼 보이기때문인데 이러한 복사는 같은 크기의 더미에 관해서 $\Theta(N)$시간 걸린다. 이 리유로 효과적인 병합을 진행하는 모든 개선된 자료구조들은 련결자료구조를 리용할것을 요구한다. 실천적으로 련결자료구조를 리용하면 다른 모든 연산들은 더 느리게 될것이다.

2진더미에서처럼 **왼쪽더미**(*leftist heap*)는 구조속성과 순서속성을 다 가진다. 사실상 왼쪽더미는 앞에서 실제로 리용된 모든 더미들에서와 같은 더미순서속성을 가진다. 더 나아가서 왼쪽더미는 2진더미이다. 왼쪽더미와 2진나무사이의 유일한 차이는 왼쪽더미가 완전한 균형이 아니라 실제적으로는 매우 불균형적이라는것이다.

1. 왼쪽더미속성

임의의 매듭 X의 **빈 경로길이**(*null path length*) $npl(X)$는 X로부터 두개의 자식을 가지지 않는 매듭까지의 가장 짧은 경로의 길이라고 정의한다. 따라서 0 또는 한개의 자식을 가진 어떤 매듭의 npl은 $npl(\mathrm{NULL})=-1$인 조건하에서 0이다. 그림 6-16에 있는 나무에서 빈 경로길이들이 나무매듭안에 표시되였다.

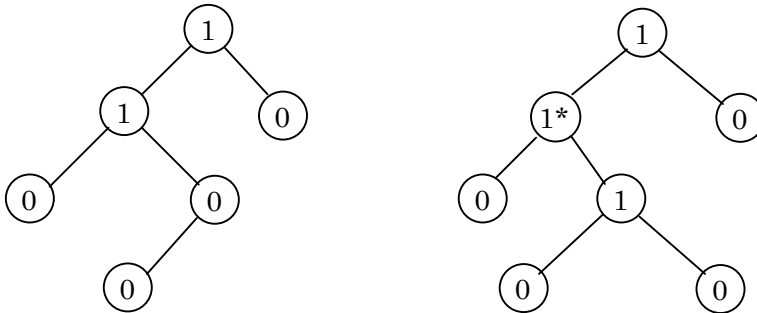



그림 6-16. 두개의 나무에 대한 빈 경로길이(왼쪽 나무만 왼쪽더미이다.)

임의의 매듭에 대한 빈 경로길이는 그 자식들의 빈 경로길이의 최소값보다 하나 더 크다. 이것은 NULL의 빈 경로길이는 -1이기때문에 두개보다 작은 자식을 가지는 매듭들에 적용된다.

왼쪽더미는 더미내에 있는 모든 매듭 X에 대하여 왼쪽 자식의 빈 경로길이가 최소한 오른쪽 자식의 빈 경로길이만큼 길다는 속성을 가지고 있다. 이 속성은 그림 6-16의 나

무들가운데서 오직 하나 왼쪽 나무에 대해서만 만족된다. 이 속성은 실제적으로 나무가 불균형이라는것을 확인하는 일은 없애 버린다. 왜냐하면 그것은 명백히 그 나무가 왼쪽으로 깊이 들어 가서 한쪽으로 기울어 지기때문이다. 사실 매듭들이 왼쪽으로 긴 경로를 이루는 나무는 현실적으로 있을수 있다(그리고 오히려 병합을 쉽게 한다.). 때문에 왼쪽더미라고 한다.

왼쪽더미는 왼쪽 경로들을 깊게 하려는 경향이 있기때문에 오른쪽 경로는 짧아 지게 된다. 사실상 왼쪽더미의 오른쪽 경로는 그 더미의 임의의 경로보다 짧다. 만일 그렇지 않으면 어떤 매듭 X를 지나서 왼쪽 자식을 가지는 경로가 있게 된다. 그러면 X는 왼쪽더미속성을 잃게 된다.

정리 6-2.

오른쪽 경로우에 r개의 매듭을 가지는 왼쪽나무는 적어도 2^r-1개의 매듭을 가져야 한다.

증명:

귀납법으로 증명하자. 만일 $r=1$이라면 적어도 한개의 나무매듭이 있어야 한다. 만일 $r\neq1$인 때 이 정리가 1, 2, 3, …, r에 대하여 참이라고 하자. 오른쪽 경로상에 $r+1$개의 매듭들을 가진 왼쪽나무를 고찰하자. 그때 뿌리는 오른쪽 경로상에 r개의 매듭들을 가지는 오른쪽 부분나무와 오른쪽 경로상에 적어도 r개의 매듭들을 가지는 왼쪽 부분나무를 가진다(만일 그렇지 않으면 그것은 왼쪽나무가 아니다.). 이 부분나무들에 귀납법의 가정을 적용하면 매 부분나무에 최소 2^r-1개의 매듭들을 준다. 이것은 뿌리를 더하여 나무에 적어도 $2^{r+1}-1$개의 매듭을 준다. 따라서 정리가 증명되였다.

이 정리로부터 N개의 매듭을 가지는 왼쪽나무는 많아서 $\lfloor \log(N+1) \rfloor$개의 매듭들을 포함하는 오른쪽 경로를 가진다는것을 쉽게 알수 있다.

왼쪽더미연산들에 대한 일반적인 방법은 오른쪽 경로상에서 모든 처리를 수행하는 것인데 그 처리는 간단히 수행된다. 오직 까다로운 부분은 오른쪽 경로상에서 insert연산과 merge연산을 실행할 때 왼쪽더미속성을 파괴할수 있다는것이다. 그것은 결국 그 속성을 회복하는것이 매우 쉬운것으로 된다.

2. 왼쪽더미연산

왼쪽더미에서 기본연산은 병합이다. 삽입은 순수 병합의 특수경우인데 그것은 어떤 삽입이 더 큰 더미를 가지는 한개 매듭더미의 merge로 고찰되기때문이다. 먼저 간단히 재귀처리를 하고 다음 이것이 어떻게 비재귀적으로 수행되는가를 보자. 입력은 두개의

왼쪽더미들인 H_1과 H_2이다. H_1과 H_2를 그림 6-17에 보여 주었다. 여기에서는 이 더미들이 실지 왼쪽더미인가를 검사해야 한다. 제일 작은 원소는 뿌리에 있다는것을 주의해야한다. 그 자료와 왼쪽과 오른쪽 지적자들에 대한 공백들외에 매 매듭은 빈 경로길이를 표시하는 하나의 입력점을 가진다.

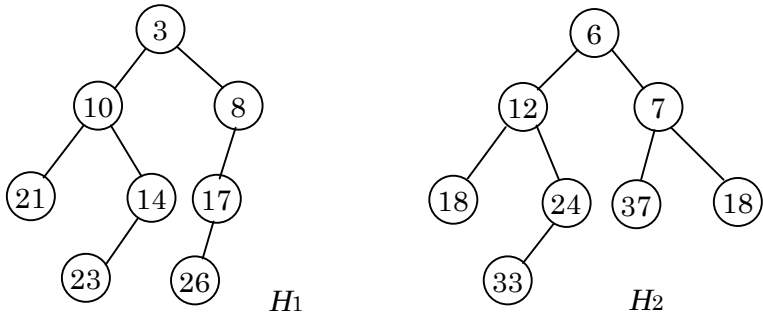



그림 6-17. 두개의 왼쪽더미 H_1과 H_2

만일 두개의 더미중에 어느 한쪽이 비면 다른 더미로 되돌아 갈수 있다. 만일 그렇지 않으면 두개의 더미를 병합하기 위해서 그 뿌리들을 비교한다. 먼저 더 큰 뿌리를 가진 더미를 더 작은 뿌리를 가진 더미의 오른쪽 부분더미와 재귀적으로 병합한다. 이것은 실례에서 H_2를 8에 뿌리를 둔 H_1의 부분더미와 재귀적으로 병합한다는것을 의미한다. 따라서 그림 6-18에 있는 더미가 얻어 진다.

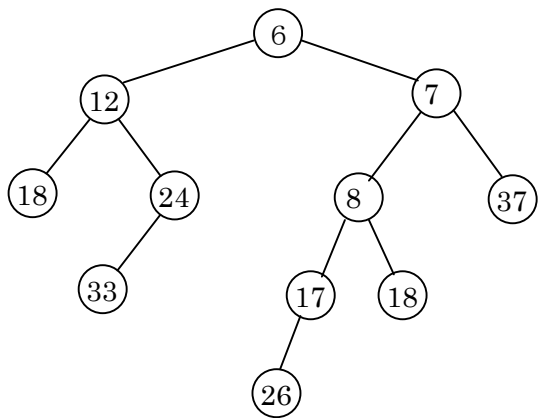



그림 6-18. H_2를 H_1의 오른쪽 부분더미와 병합한 결과

이 나무가 재귀적으로 만들어 지고 그리고 아직 그 알고리듬서술이 끝나지 않았기때문에 이 시점에서 이 더미가 어떻게 얻어 지는가를 보여 줄수 없다. 이 시점에서 이 더미가 초래한 나무가 왼쪽더미라고 가정하는것이 타당하다. 왜냐하면 그 나무가 재귀적인 처리로 얻어 졌기때문이다. 이것은 귀납법에 의한 증명에서 귀납적가정과 아주 류사하다. 기초적인 경우(한개의 나무가 빌 때 발생하는)를 처리할수 있으므로 재귀단계는 병합을

완료할 때까지 처리를 수행한다고 가정할수 있다.

이것은 재귀의 세번째 규칙인데 이에 대하여서는 제1장에서 설명하였다. 이제 이 새로운 더미를 H_1의 뿌리의 오른쪽 자식으로 만든다(그림 6-19).

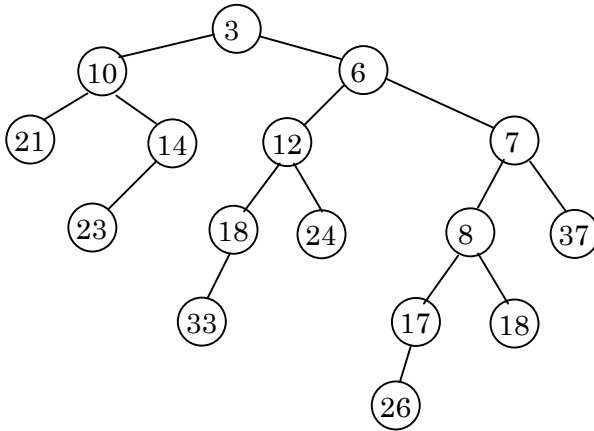



그림 6-19. 앞의 그림의 왼쪽 더미를 H_1의 오른쪽 자식으로 붙인 결과

비록 결과적인 더미가 더미순서속성을 만족한다고 하더라도 뿌리의 왼쪽 부분나무가 빈 경로길이 2를 가지는데 도리여 왼쪽 부분나무는 빈 경로길이 1을 가지기때문에 왼쪽나무가 아니다. 따라서 뿌리에서 왼쪽더미속성이 위반된다. 그러나 그 나무의 나머지는 왼쪽더미라는것을 쉽게 알수 있다. 그 재귀단계로 하여 뿌리의 왼쪽 부분나무는 왼쪽더미이다. 그 뿌리의 왼쪽 부분나무는 변화되지 않는데 그래서 그것은 여전히 왼쪽더미이여야 한다. 따라서 오직 그 뿌리만을 고정할 필요가 있다. 간단히 뿌리의 왼쪽과 오른쪽 자식들을 치환하고(그림 6-20) 빈 경로길이(그 새로운 빈 경로길이는 1에 새로운 오른쪽 자식의 빈 경로길이를 더한것)를 갱신하여 전체 나무를 왼쪽더미로 만들수 있다. 이것으로 merge를 완료한다. 만약 빈 경로길이가 갱신되지 않았다면 모든 빈 경로길이는 0일것이며 그 더미는 왼쪽더미는 아니지만 순수 우연적이라는데 주의를 돌려야 한다. 이 경우에 알고리듬은 동작하지만 요구한 그 시간한계는 그보다 더 떨어 질것이다.

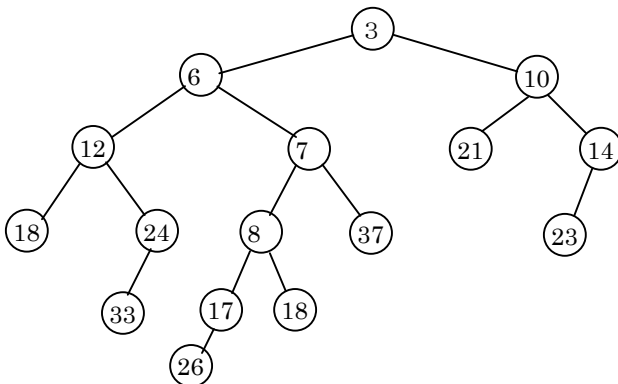



그림 6-20. H_1 의 뿌리의 자식들을 치환한 결과

그 알고리듬은 코드로 직접 변환된다. 매듭클라스(프로그람 6-5)는 npl(빈 경로길이) 자료성원으로 추가된다는것을 제외하고 2진나무에서와 같다.

왼쪽더미는 뿌리에 대한 지적자를 자료성원으로서 기억한다. 제4장에서 어떤 요소가 빈 2진나무에 삽입될 때 그 뿌리에 의해서 참조되는 매듭은 변화되여야 한다는것을 보았다. 병합을 처리하기 위하여 private재귀산법들을 실행하는 일반적인 기법을 리용한다. 그 클라스구조를 프로그람 6-5에 보여 주었다.

```
template <class Comparable>
class LeftistHeap;
template <class Comparable>
class LeftistNode
{
    Comparable     element;
    LeftistNode *left;
    LeftistNode *right;
    int npl;
    LeftistNode( const Comparable & theElement, LeftistNode *lt = NULL,
                  LeftistNode *rt = NULL.    int np = 0 )
      : element( theElement ), left( lt ), right( rt ), npl( np ) { }
    friend class LeftistHeap<Comparable>;
};
template <class Comparable>
class LeftistHeap
{
   public:
    LeftistHeap( );
    LeftistHeap( const LeftistHeap & rhs );
    ~LeftistHeap( );
   bool isEmpty( ) const;
   bool isFull( ) const;
   const Comparable & findMin( ) const;
   void insert( const Comparable & x );
   void deleteMin( );
   void deleteMin( Comparable & minltem );
   void makeEmpty( );
   void merge( LeftistHeap & rhs );
   const LeftistHeap & operator=( const LeftistHeap & rhs );
 private:
   LeftistNode<Comparable> *root;
   LeftistNode<Comparable>* merge( LeftistNode<Comparable> *h1,
                                   LeftistNode<Comparable> *h2 ) const;
   LeftistNode<Comparable> * mergel( LeftistNode<Comparable> *h1,
                                     LeftistNode<Comparable> *h2 ) const;
 void swapChildren( LeftistNode<Comparable> * t ) const;
   void reclaimMemory( LeftistNode<Comparable> * t ) const;
```

```
    LeftistNode<Comparable> * clone( LeftistNode<Comparable> *t ) const;
};
```

프로그람 6-5. 왼쪽더미의 형선언

두개의 merge루틴(프로그람 6-6)들은 특수한 경우를 제외하고 H_1이 더 작은 뿌리를 가지도록 설계되였다. 실제적인 병합은 mergel에서 실현된다(프로그람 6-7). 공개부의 merge방법에서 rhs를 그 조종더미에 병합한다. rhs는 비게 된다. 공개부의 방법에서 가명검사는 h.merge(h)를 허용하지 않는다.

```
/**
* Merge rhs into the priority queue.
*   rhs becomes empty, rhs must be different from *this.
*/
template <class Comparable>
void LeftistHeap<Comparable>: :merge( LeftistHeap & rhs )
{
    if( this == &rhs )        // Avoid aliasing problems
        return;
    root = merge( root, rhs.root );
    rhs.root = NULL;
}
/**
* Internal method to merge two roots.
* Deals with deviant cases and calls recursive mergel.
*/
template <class Comparable>
LeftistNode<Comparable> *
LeftistHeap<Comparable>: :merge( LeftistNode<Comparable> * h1,
                                  LeftistNode<Comparable> * h2 ) const
{
     if( h1 == NULL )
         return h2;
     if( h2 == NULL )
         return h1;
     if( hl->element < h2->element )
         return merge1( h1, h2 );
     else
         return merge1( h2, h1 );
     }
```

프로그람 6-6. 왼쪽더미들을 병합하기 위한 구동루틴들

```
/**
* Internal method to merge two roots.
* Assumes trees are not empty, and h1's root contains smallest item.
```

```
*/
template <class Comparable>
LeftistNode<Comparable>*
LeftistHeap<Comparable>::mergel( LeftistNode<Comparable> * h1,
                                 LeftistNode<Comparab1e> * h2 ) const
{
    if( h1->left == NULL )      // Single node
          h1->left = h2;                    // Other fields in h1 are already accurate
    Else
    {
       h1->right = merge( h1->right, h2 );
       if( h1->left->npl < h1->right->npl )
            swapChildren( h1 );
       h1->npl = h1->right->npl + 1;
    }
    return h1;
}
```

프로그램 6-7. 왼쪽더미의 병합을 위한 실제루틴

병합을 실현하는 시간은 오른쪽 경로길이들의 합에 비례하는데 그것은 재귀호출시 거치는 매개 매듭에서 상수적인 처리가 진행되기때문이다. 따라서 두 왼쪽더미들을 병합하는데 $O(logN)$시간한계를 얻는다. 또한 이 연산을 본질적으로 두 단계로 실행함으로써 비재귀적으로 수행할수도 있다. 첫번째 단계에서는 량쪽 더미의 오른쪽 경로들을 병합하는 방법으로 새로운 나무를 만든다. 이를 위하여 그것들의 매 왼쪽 자식들을 유지하면서 H_1과 H_2의 오른쪽 경로상에 있는 매듭들을 정렬된 순서로 정돈한다. 실례에서 새로운 오른쪽 경로는 3, 6, 7, 8, 18이며 그 결과적인 나무를 그림 6-21에 보여 주었다. 두번째 단계는 그 더미에서 진행되며 왼쪽더미속성을 위반하는 매듭들에서 자식의 치환이 실행된다. 그림 6-21에서 매듭 7과 3이 치환되여 그림 6-21과 같은 나무가 얻어 진다.

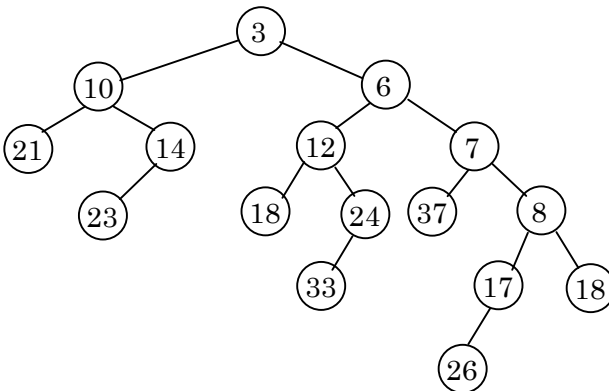



그림 6-21. H_1과 H_2의 오른쪽 경로들을 병합한 결과

비재귀의 변종을 시각적으로 보는것은 간단하지만 코드화는 더 어렵다. 재귀와 비재귀수속들이 같은 처리를 진행한다는것을 독자들의 리해에 맡긴다.

우에서 언급된것처럼 어떤 항목을 한매듭더미에 삽입하고 merge연산을 한번 실행함으로써 삽입을 수행할수 있다. deleteMin연산을 수행하기 위하여 병합될수 있는 두개의 더미를 만들어 간단히 그 뿌리를 해제한다. 따라서 deleteMin연산을 수행하는 시간은 $O(logN)$이다. 이 두개의 루틴은 프로그람 6-8과 6-9에 코드화되였다.

```
/**
* Insert item x into the priority queue, maintaining heap order
*/
template <class Comparable>
void LeftistHeap<Comparable>: :insert( const Comparable & x )
{
    root = merge( new LeftistNode<Comparable>( x ), root );
}
```

프로그람 6-8. 왼쪽더미들을 위한 삽입루틴

```
/**
 * Remove the smallest item from the priority queue.
 * Throws Underflow if empty.
 */
template <class Comparable>
void LeftistHeap<Comparable>: :deleteMin( )
{
       if( isEmpty( ) )
          throw Underflow( );
       LeftistNode<Comparable> "oldRoot = root;
       root = merge( root->left, root->right );
       delete oldRoot;;
}
```

프로그람 6-9. 왼쪽더미들을 위한 deleteMin 루틴

마지막으로 2진더미를 가지고 $O(N)$시간에 왼쪽더미를 만들수 있다(명백하게 련결실현을 리용하여). 비록 2진더미가 정확히 왼쪽더미라고 하여도 이것은 반드시 제일 좋은 풀이가 아닌데 그것은 얻어 진 더미가 가능한 최악의 왼쪽더미이기때문이다. 더우기 역준위순서로 나무를 순회하는것은 련결구조들에 관해서 쉽지는 않다. BuildHeap연산의 효과는 재귀적으로 왼쪽과 오른쪽 부분나무들을 만들고 다음에 그 뿌리를 아래로 려과함으로써 얻어 질수 있다. 련습에서는 또 다른 풀기방법을 제시한다.

제7절. 경사더미

경사더미(*skew heap*)는 아주 간단히 실행되는 어떤 왼쪽더미에 대한 자체조정구조의 한가지 형태이다. 경사더미와 왼쪽더미와의 관계는 **펼친**(*splay*)나무와 AVL나무사이의 관계와 류사하다. 경사더미들은 더미순서속성을 가진 2진나무들이지만 이 나무들에는 어떤 구조적인 제한도 없다. 왼쪽더미들과는 달리 임의의 매듭의 빈 경로길이에 대한 정보는 보존되지 않는다. 경사더미의 오른쪽 경로는 어느때든지 임의로 길어 질수 있으며 따라서 모든 연산들에 대한 최악의 경우의 실행시간은 $O(N)$이다. 더우기 펼친나무들에서처럼 어떤 M개의 련속적인 연산들에 대하여 최악의 경우 전체 실행시간은 $O(MlogN)$이다(제11장을 참고). 따라서 경사더미들은 매 연산당 비용으로 보상된 $O(logN)$을 가진다. 왼쪽더미에서처럼 경사더미에서의 기본연산은 병합이다. merge루틴은 또다시 재귀이며 한가지를 제외하고는 앞에서와 같은 연산들을 정확히 수행한다. 그 차이는 왼쪽더미들에서 왼쪽과 오른쪽 자식들이 왼쪽더미구조속성을 만족하는가 안하는가를 검사하고 만약 만족하지 않는다면 그것들을 교환하는것이다. 경사더미들에서 교환은 비조건적이다. 즉 오른쪽 경로우에 있는 모든 매듭들가운데서 가장 큰 매듭은 그의 자식들을 교환하지 않는다는것을 제외하고는 언제나 교환을 진행한다. 이 한가지 제한은 자연적인 재귀실행에서 발생하는것이며 따라서 그것은 실제로 조금도 특수한 경우는 아니다. 더우기 그 한계를 확증할 필요는 없지만 이 매듭은 오른쪽 자식을 가지지 못하게끔 되여 있기때문에 교환을 진행하고 그것을 매듭에 주는것은 필요 없는 일이다(이 실례에서는 이 매듭의 자식들이 없기때문에 그에 대하여 주의할 필요는 없다.). 또다시 입력이 앞에서와 같이 두개의 더미라고 하자(그림 6-22).

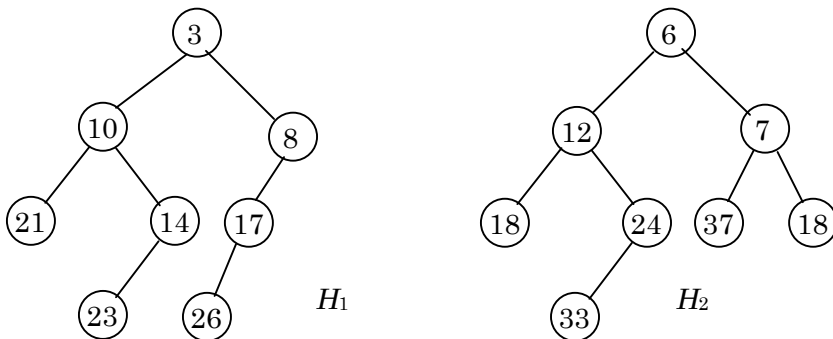



그림 6-22. 2개의 경사더미 H_1과 H_2

만약 H_2를 8에 뿌리를 둔 H_1의 부분더미와 재귀적으로 병합한다면 그림 6-23과 같은 더미를 얻는다.

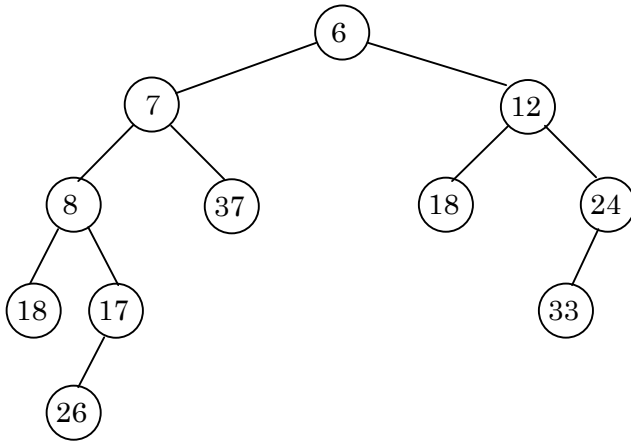



그림 6-23. H_2를 H_1의 오른쪽 부분더미와 병합한 결과

이것은 다시 재귀적으로 실행되며 따라서 재귀의 세번째 규칙(제1장 제3절)에 의해 그것을 얻는 방법에 대하여 걱정할 필요는 없다. 이 더미는 왼쪽더미이지만 이것은 언제나 왼쪽더미로 된다고 장담하지는 못한다. 이 더미를 H_1의 새로운 왼쪽 자식으로 만들고 H_1의 본래의 왼쪽 자식은 새로운 오른쪽 자식으로 된다(그림 6-24).

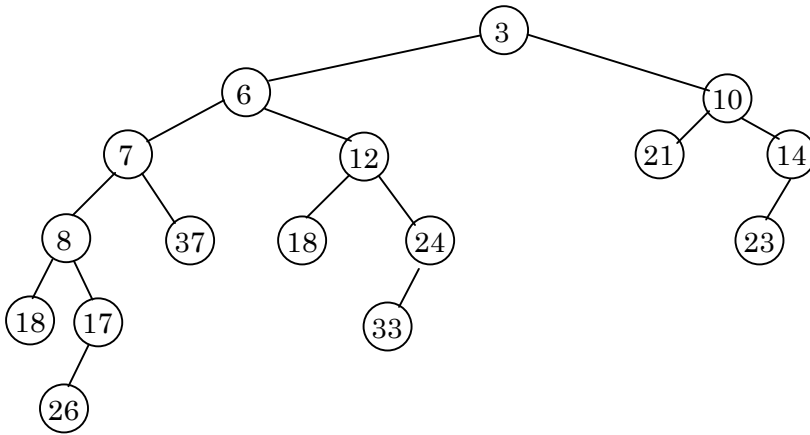



그림 6-24. 경사더미 H_2와 H_2을 병합한 결과

전체 나무는 왼쪽더미이지만 그것은 언제나 그렇게 되지 않는다는것을 쉽게 알수 있다. 즉 15를 이 새로운 더미에 삽입하면 왼쪽더미속성이 파괴될것이다.

왼쪽더미에서처럼 오른쪽 경로들을 병합한 다음 마지막매듭을 제외하고 오른쪽 경로우에 있는 매개 매듭들에 대하여 왼쪽과 오른쪽 자식들을 교환함으로써 모든 연산들을 비재귀적으로 실행할수 있다. 몇개의 실례들을 고찰하여 보면 거의 다름없이 오른쪽 경로우에 있는 마지막매듭은 교환된 자식들을 가지기때문에 총적인 결과는 이것이 새로운 왼쪽 경로로 된다는것이다(이것을 확인하려면 앞의 실례를 보시오.). 이것은 두개의 경

사다미들을 매우 쉽게 병합하게 한다.[22] 경사더미의 실행은 련습으로 좀 남겨 두자. 오른쪽 경로가 길수 있기때문에 가령 실행은 다른 방식으로 접수된다하더라도 재귀실행은 탄창공간의 부족으로 하여 실패할수 있다는것을 주의해야 한다. 경사더미는 경로의 길이를 보존할 여분의 공간을 요구하기때문에 자손을 치환하겠는가를 결정하는 어떤 검사도 하지 않는다는 우점을 가진다. 그것은 왼쪽더미와 경사더미에 대해 정확히 기대되는 오른쪽 경로길이를 결정하기 위한 열린문제이다(후자는 확실히 더 어렵다.). 이와 같은 비교는 균형정보의 적은 손실을 검사부족으로 상쇄되는가를 쉽게 결정할수 있게 한다.

제8절. 2항대기렬

왼쪽더미와 경사더미는 둘 다 병합, 삽입, deleteMin연산들을 모두 연산당 $O(logN)$ 시간동안에 능률적으로 실행한다고 해도 2진더미들이 연산당 상수평균시간에 삽입연산을 처리하기때문에 그 시간한계를 개선할 여지가 많다. 2항대기렬은 이 3개의 연산들을 최악의 경우 연산당 $O(logN)$의 시간으로 지원한다. 그러나 삽입은 평균적으로 상수시간에 처리된다.

1. 2 항대기렬구조

2항대기렬(*binomial queue*)들은 더미순서나무는 아니지만 **수림**(f*orest*)이라고 하는 더미순서나무들의 집합이라는 점에서 모든 우선권대기렬실현들과 같지 않다. 매개 더미순서나무들은 **2항나무**(*binomial tree*)로 알려 진 제한된 형태들중의 하나이다(2항나무에 대한 이름의 의미는 후에 명백히 고찰한다.). 각이한 모든 높이를 가지는 2항나무는 기껏해서 하나 있다. 높이가 령인 2항나무는 한매듭나무이며 높이가 k인 2항나무 B_k는 2항나무 B_{k-1}을 다른 하나의 2항나무 B_{k-1}의 뿌리에 붙여 만든다. 그림 6-25에서는 2항나무B_0, B_1, B_2, B_3, B_4를 보여 주었다.

그림으로부터 2항나무 B_k는 자식 B_0 ,B_1 ,…,B_{k-1}들을 가지는 뿌리로 구성된다는것을 알수 있다. 높이가 k인 2항나무는 정확히 2^k개의 매듭을 가지며 깊이 d에서 매듭들의 수는 2항결수 $\binom{k}{d}$이다. 만약 2항나무들에 대하여 더미순서를 강요하고 어떤 높이를 가지는 2항나무를 많아서 하나 허용한다면 우리는 임의의 크기의 우선권대기렬을 2항나무들의 집합으로 유용하게 표현할수 있다. 실례로 크기가 13인 우선권대기렬은 수림 B_3, B_2, B_0으로

[22] 이것은 정확히 재귀실행과 같지 않지만 같은 시간한계를 산출한다. 만약 오른쪽 경로상에 있는 매듭들에 대해서만 자식들을 치환한다면 재귀변종과 같은 결과를 얻는다. 여기서 그 자식들은 한개 더미의 오른쪽 경로가 비여 있기때문에 오른쪽 경로들의 병합이 끝났다는것을 알수 있다.

표현할수 있다. 이 표현을 1101로 쓸수 있는데 이것은 10진수로 13을 나타낼뿐아니라 B_3, B_2, B_0은 2항대기렬이 존재하고 B_1은 존재하지 않는다는 사실을 나타낸다.

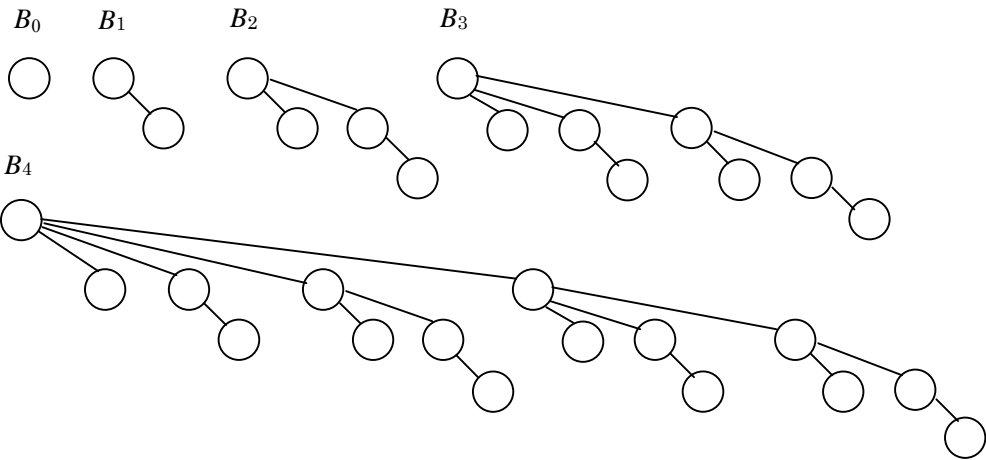
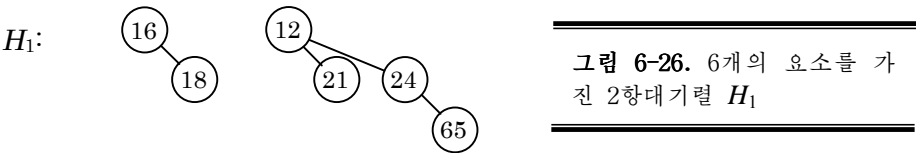



그림 6-25. 2항나무 B_0, B_1, B_2, B_3, B_4

실례로 6개의 요소를 가지는 우선권대기렬을 그림 6-26에서와 같이 표현할수 있다.

그림 6-26. 6개의 요소를 가진 2항대기렬 H_1

2. 2항대기렬연산

최소값요소는 모든 나무들의 뿌리를 조사하여 찾을수 있다. 기껏해서 log*N*개의 각이한 나무들이 있으므로 최소값을 $O(logN)$시간에 찾을수 있다. 또 다른 방도은 최소값이 다른 연산과정에 변화될 때 제일 최신의것으로 갱신된다는것을 기억한다면 최소값을 보전하고 그 연산을 $O(1)$시간에 실행할수 있다.

두개의 2항대기렬을 병합하는것은 개념적으로 쉬운 연산이다. 그 연산을 실례를 들어 고찰하자. 각각 6개, 7개의 요소를 가진 2항대기렬 H_1와 H_2있 있다고 하자. 그것을 각각 그림 6-27에서 보여 주었다.

병합은 사실상 두개의 대기렬들을 함께 결합하여 실행한다. H_3은 새로운 2항대기렬이라고 하자. H_1은 높이가 0인 2항나무를 가지지 않으며 H_2는 높이가 0인 2항나무를 가

진다는데로부터 H_2에 있는 높이가 0인 2항나무를 H_3의 부분나무로 리용할수 있다. 다음 높이가 1인 2항나무를 더한다. H_1과 H_2은 둘다 높이가 1인 2항나무를 가지기때문에 더 큰 뿌리를 가진 나무를 더 작은 뿌리를 가진 나무의 부분나무로 만들어 높이가 2인 2항나무를 만드는것으로써 그것들을 병합한다(그림6-28). 따라서 H_3은 높이가 1인 2항나무를 가지지 않는다. 현재 높이가 2인 3개의 2항나무(다시말해서 초기의 H_1과 H_2의 나무들에 앞단계에 의해 형성된 나무를 더하여)가 있다. H_3에 높이가 2인 하나의 2항나무를 보존하고 높이가 3인 2항나무를 만들면서 다른 2개를 병합한다. H_1과 H_2은 높이가 3인 나무를 가지지 않는다는데로부터 이 나무는 H_3의 부분으로 되여 처리는 끝난다. 결과적인 2항대기렬을 그림 6-29에 보여 주었다.

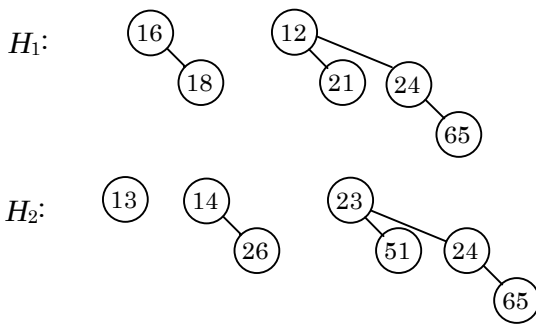



그림 6-27. 2개의 2항대기렬 H_1와 H_2

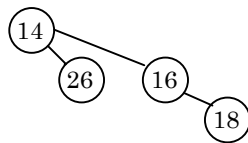



그림 6-28. H_1와 H_2내의 두개의 B_1나무들의 병합

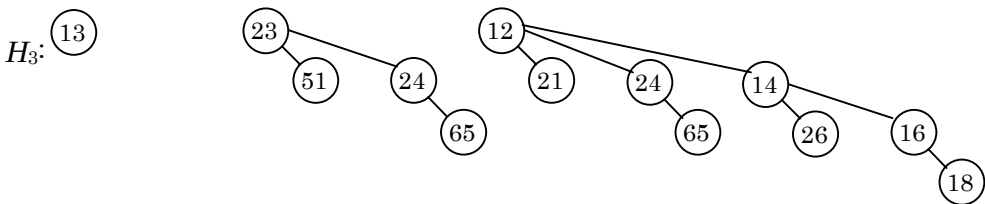



그림 6-29. H_1과 H_2를 병합한 2항대기렬 H_3

두개의 2항나무들의 병합은 대부분의 어떤 합리적인 실현에 상수시간을 가지며 $O(logN)$개의 2항나무들이 있기때문에 병합은 최악의 경우 $O(logN)$시간이 걸린다. 이 연산을 효과적으로 하자면 그 나무들을 높이에 의해서 정렬된 2항대기렬로 유지하여야 하는데 그것은 물론 간단히 처리된다.

삽입은 바로 병합의 특수한 경우인데 그것은 간단히 한매듭나무를 생성하고 병합을 실행하기때문이다. 최악의 경우 이 연산시간은 역시 $O(logN)$로 된다. 더 정확히 말하여 만일 요소가 존재하는 곳에 삽입되는 우선권대기렬이 가장 작은 비존재2항나무가 B_i이라는 속성을 가진다면 그 실행시간은 i+1에 비례한다. 실례로 H_3(그림6-29)에서는 높이가 1인 2항나무를 놓쳤으며 따라서 삽입은 2개의 단계로 끝나게 된다. 2항대기렬의 매 나무는 $\frac{1}{2}$의 확률을 가지기때문에 삽입을 두 단계로 끝내도록 기대하며 따라서 평균시간은 변하지 않는다. 더우기 초기에 빈 2항대기렬에 대한 N개의 insert연산들의 실행시간은 최악의 경우 실행시간은 $O(N)$이다. 사실 이 연산은 오직 N-1번의 비교만으로 가능한데 이것을 련습문제로 남긴다.

실례로 그림 6-30에서부터 그림 6-38까지에서는 1~7까지 차례로 삽입하여 만들어 지는 2항대기렬을 보여 주었다. 4를 삽입하는것은 나쁜 경우를 보여 준다. 4를 B_0과 병합하여 높이가 1인 새로운 나무를 얻는다. 다음 이 나무를 B_1과 병합하여 높이가 2인 나무를 얻는데 이 나무는 새로운 우선권대기렬이다. 이것을 세단계로 계수한다(두개의 나무의 병합에 정지경우를 더하여). 7이 삽입된 다음의 삽입은 또 하나의 나쁜 경우이며 3개의 나무병합을 요구한다.

deleteMin연산은 먼저 제일 작은 뿌리를 가지는 2항나무를 찾는 방법으로 수행된다. 이 나무를 B_k, 그리고 초기의 우선권대기렬을 H라고 하자. H내에 있는 나무들을 가지는 수림으로부터 2항나무 B를 삭제하여 새로운 2항대기렬 H'를 형성한다. 또한 2항나무들 $B_0, B_1, \cdots, B_{k-1}$을 생성하여 B_k의 뿌리를 삭제한다. 따라서 결과적으로 우선권대기렬 H''를 형성한다. 그리고 H'와 H''를 병합하여 그 연산을 끝낸다.

실례에서처럼 H_3에 대한 deleteMin연산(그림 6-37에서 다시 보여 준다.)을 수행한다고 하자. 최소값뿌리는 12이며 따라서 그림 6-38과 6-39에 있는 두개의 우선권대기렬 H_1과 H_2를 얻는다. H'와 H''의 병합으로부터 얻어 지는 2항대기렬은 마지막결과로서 그림 6-40에 보여 주었다.

그림 6-30. 1이 삽입된후

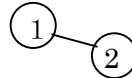



그림 6-31. 2가 삽입된후

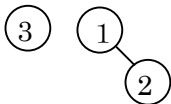



그림 6-32. 3이 삽입된후

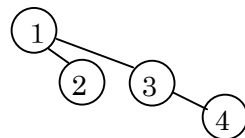



그림 6-33. 4가 삽입된후

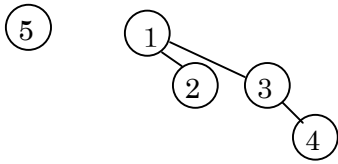



그림 6-34. 5가 삽입된후

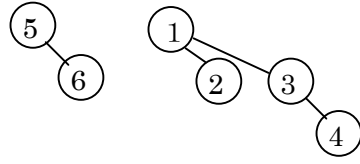



그림 6-35. 6이 삽입된후

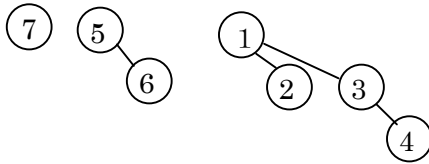



그림 6-36. 7이 삽입된후

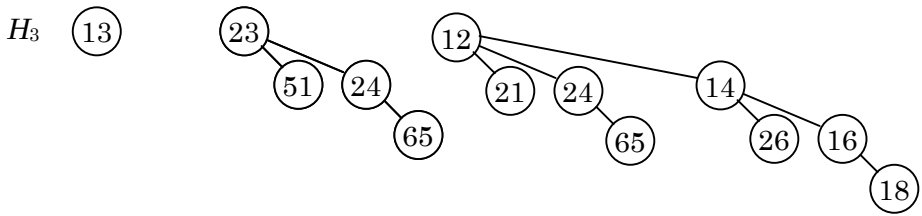



그림 6-37. 2 항대기렬 H_3

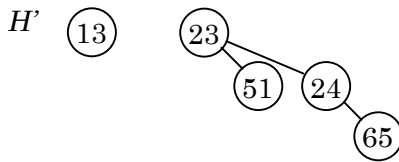



그림 6-38. B_3을 제외하고 H_3에 있는 모든 2항나무들을 포함하는 2항대기렬 H'

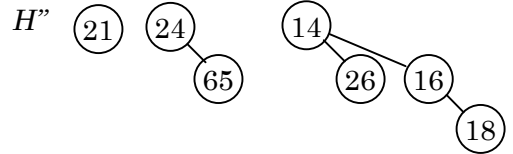



그림 6-39. 2항대기렬 H'': 12가 제거된 다음의 B_3

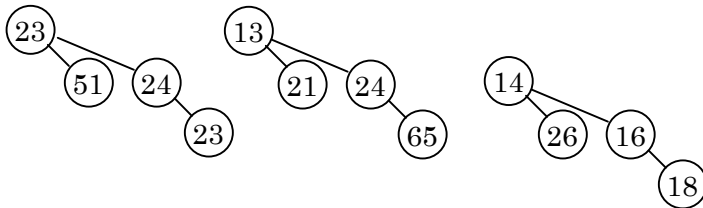



그림 6-40. H_3에 deletemin 연산을 적용한 결과

이것을 분석을 위해서 먼저 deleteMin연산은 본래의 2항대기렬을 둘로 가른다는것에 주의하시오. 최소값요소를 포함하는 나무를 찾고 대기렬들 H'와 H''를 만드는데 $O(logN)$시간이 걸린다. 이 두 개의 병합은 $O(logN)$시간이 걸리며 따라서 전체 deleteMin 연산은 $O(logN)$시간이 걸린다.

3. 2항대기렬의 실현

deleteMin연산은 뿌리의 모든 부분나무들을 고속으로 탐색할것을 요구하며 따라서 일반나무를 표준형태로 만들어야 한다. 즉 매 매듭의 자식들은 련결목록에 보존되고 매개 매듭은 그의 첫번째 자식에 대한 지적자를 가진다. 또한 이 연산에서는 그 자식들이 그의 부분나무들의 크기에 따라 순서화되여 있어야 한다. 두개의 나무를 병합하는것은 쉽다. 두개의 나무들이 병합될 때 그 나무들중의 하나는 다른 나무에 대한 자식으로서 추가된다. 이 새로운 나무는 가장 큰 부분나무이므로 그 부분나무들을 크기가 작아 지는 순서로 유지되여야 한다. 그러면 오직 2개의 2항나무만을 병합할수 있으며 그에 따라서 2항대기렬들도 효과적으로 병합된다. 2항대기렬들은 2항나무들의 배렬이다.
간단히 말하여 2항나무의 매개 매듭은 자료와 첫번째 자식,그리고 오른쪽 형제를 포함한다. 2항나무의 자식들은 작아 지는 위수로 정돈된다. 그림6-42는 그림 6-41에 있는 2항대기렬을 표현하는 방법을 보여 준다. 프로그람 6-10은 2항나무의 매듭에 대한 형선언과 2항대기렬클라스대면부를 보여 준다.

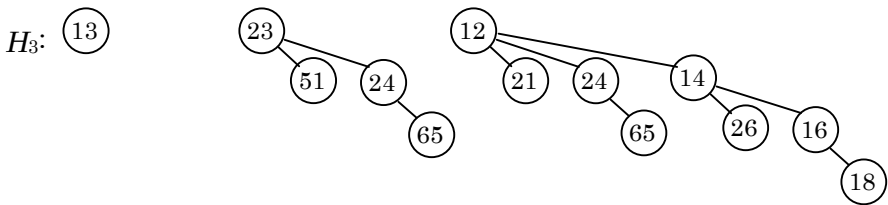



그림 6-41. 수림으로 표시한 2항대기렬 H_3

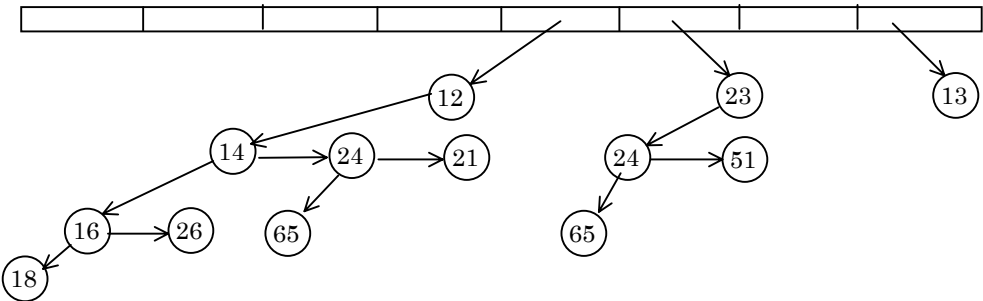



그림 6-42. 2항대기렬 H_3의 표현

```
template <class Comparable>
class Binomial Queue;
template <class Comparable>
class BinomialNode
{
    Comparable       element;
    BinomialNode *leftChild;
    BinomialNode *nextSibling;
    BinomialNode( const Comparable & theElement,
                  BinomialNode *lt, Binomial Node *nt )
: element( theElement ), leftChild( lt ), nextSib1ing( nt ) { }
    friend class BinomialQueue<Comparable>;
};
template <class Comparable>
class BinomialQueue
{
    public:
      Binomial Queue( );
      BinomialQueue( const BinomialQueue & rhs );
      ~BinomialQueue( );
      bool isEmpty( ) const;
      bool isFull( ) const;
      const Comparable & findMin( ) const;
      void   insert( const Comparable & x );
      void   deleteMin( );
      void   deleteMin( Comparable & minItem );
      void   makeEmpty( );
      void   merge( BinomialQueue & rhs );
      const BinomialQueue & operator=( const BinomialQueue & rhs );
private:
   int currentSize;              // Number of items in the priority queue
   vector<BinomialNode<Comparable> *> theTrees; // An array of tree roots

   int findMinIndex( ) const;
   int capacity( ) const;
   BinomialNode<Comparable> *combineTrees( BinomialNode<Comparable> *t1,
                                           BinomialNode<Comparable> *t2 ) const;
   void makeEmpty( BinomialNode<Comparable> * & t ) const;
   BinomialNode<Comparable> * clone( BinomialNode<Comparable> * t ) const;
};
```

프로그람 6-10. 2항대기렬클라스대면부와 매듭정의

두개의 2항대기렬을 병합하기 위해서 같은 크기를 가지는 두개의 2항나무들을 병합하는 루틴을 요구한다. 그림 6-43에서는 두개의 2항나무들을 병합할 때 련결을 어떻게 변화시키는가를 보여 주었다. 이것을 수행하는 코드는 단순하며 그것을 프로그람 6-11에 보여 주었다.

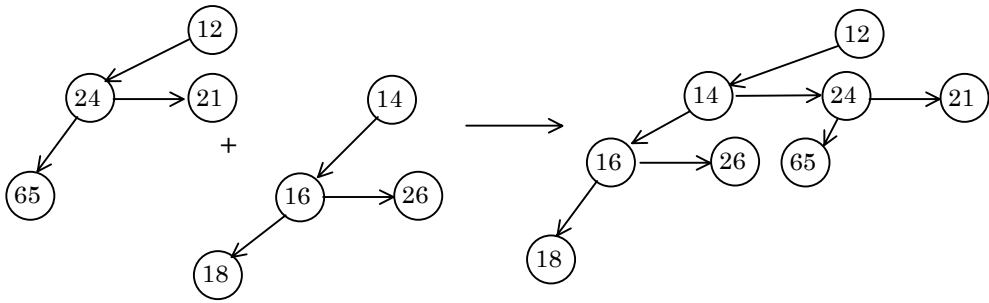



그림 6-43. 두개의 2 항나무들의 병합

```
/**
* Return the result of merging equal-sized t1 and t2.
*/
Template <class Comparable>
BinomialNode<Comparable> *
BinomialQueue<Comparable>::combineTrees(BinomiaINode<Comparable>*tl,
                                         BinomialNode<Comparable> *t2 ) const
{
      if( t2->element < t1->element )
           return combineTrees(t2, t1 );
      t2->nextSibling = t1->leftChild;
      tl->leftChild = t2;
      return t1;
}
```

프로그램 6-11. 같은 크기를 가지는 두개의 2 항나무들을 병합하는 루틴

여기에서 merge루틴의 간단한 실현을 준다. H_1은 현재 객체로 표현되며 H_2는 rhs로 표현된다. 이 루틴은 그 결과를 H_1에 넣고 H_2를 비게 함으로써 H_1과 H_2를 결합한다. 임의의 시점에서 위수 i의 나무들을 처리한다. t_1과 t_2는 각각 H_1과 H_2에 속하는 나무들이며 carry는 앞의 단계에서 만들어 진 나무이다(그것은 NULL이여도 좋다.). 8개의 가능한 매 경우에 의존하여 위수 I에서 발생한 나무와 위수 i+1에 대한 carry나무가 만들어 진다. 이 과정은 결과로 되는 2항대기렬에서 위수 0부터 마지막위수까지 처리된다. 그 코드를 프로그램 6-12에 보여 주었다.

```
/**
  * Merge rhs into the priority queue.
  * rhs becomes empty. rhs must be different from *this.
  * Throw Overflow    if result exceeds capacity.
  */
template <class Comparable>
```

```
void BinomialQueue<Comparable>: :merge( BinomialQueue<Comparable> & rhs )
{
   if( this == &rhs )        // Avoid aliasing problems
        return;
   if( currentSize + rhs.currentSize > capacity( ) )
        throw Overflow( );
   currentSize += rhs.currentSize;
   BinomialNode<Comparable> *carry = NULL;
   for( int i = 0, j = 1; j <= currentSize; i++, j *= 2 )
   {
BinomialNode<Comparable> *t1 = theTrees[ i ];
       BinomialNode<Comparable> *t2 = rhs.theTrees[ i ];
       int whichCase = t1 == NULL ? 0 : 1;
       whichCase += t2 == NULL ? 0 : 2;
       whichCase += carry == NULL ? 0 : 4;
     switch( whichCase )
     {
        case 0: /* No trees */
        case 1: /* Only *this */
             break;
        case 2: /*Only rhs */
             theTrees[ i ] = t2;
             rhs.theTrees[ i ] = NULL;
             break;
        case 4: /* Only carry */
             theTrees[ i ] = carry;
             Carry = NULL;
             break;
        case 3: /* *this and rhs */
             Carry = combineTrees( t1, t2 );
             theTrees[ i ] = rhs.theTrees[ i ] = NULL;
             break;
        case 5: /* *this and carry */
             Carry = combineTrees( t1, carry );
             theTrees[ i ] = NULL;
             break;
        case 6: /* rhs and carry */
             Carry = combineTrees( t2, carry );
             rhs.theTrees[ i ] = NULL;
             break;
        case 7: /* All three */
             theTrees[ i ] = carry;
             Carry = combineTrees( t1, t2 );
             rhs.theTrees[ i ] = NULL;
             break;
     }
   }
   for( int k = 0; k < rhs.theTrees.size( ); k++ )
             rhs.theTrees[ k ] = NULL;
```

```
 rhs.currentSize = 0;
 }
```

프로그램 6-12. 두개의 우선권대기렬을 병합하기 위한 루틴

2항대기렬에 대한 deleteMin루틴을 프로그램 6-13에 주었다.

```
/**
* Remove the smallest item from the priority queue and
* copy it into minltem. Throw Underflow if empty.
*/
template <class Comparable>
void BinomialQueue<Comparable>::deleteMin( Comparable & minltem )
{
   if( isEmpty( ) )
          throw Underflow( );
   int minlndex = findMinIndex( );
   minltem = theTrees[ minlndex ]->element;

   BinomialNode<Comparable> *oldRoot = theTrees[ minlndex ];
   BinomialNode<Comparable> *deletedTree = oldRoot->leftChild;
   delete oldRoot;
   // Construct H"
   BinomialQueue deletedQueue;
   deletedQueue.currentSize = ( 1 << minlndex ) - 1;
   for( int j = minlndex - 1; j >= 0; j - -)
   {
      deletedQueue.theTrees[ j ] = deletedTree;
      deletedTree == deletedTree->nextSibling;
      deletedQueue.theTrees[ j ]->nextSibling = NULL;
     }
     // Construct H'
theTrees[ minlndex ] = NULL;
currentSize -= deletedQueue.currentSize + 1;
merge( deletedQueue );
      }
/**
  * Find index of the tree containing the smallest item in the
  * priority queue. The priority queue must not be empty.
  * Return the index of the tree containing the smallest item.
  */
template <class Comparable>
int BinomialQueue<Comparable>: :findMinIndex( ) const
{
      int i;
      int minIndex;
      for(    I = 0; theTrees[ i ] == NULL; 1++ );
      for( minIndex = i; i < theTrees.size( ); i++ )
```

```
            if( theTrees[ i ] != NULL && theTrees[ 1 ]->element
          < theTrees[ minlndex ]->element )
          minIndex = i;
     return minIndex
  }
```

프로그램 6-13. 2항대기렬을 위한 deletemin 연산

할당된 요소들의 위치가 알려 질 때 2진더미들은 decreaseKey와 remove와 같은 일부 비표준연산들을 지원할수 있도록 2항대기렬들을 확장할수 있다. decreaseKey는 우로려파인데 이것은 부모련결을 보관하는 매개 매듭에 대한 자료성원을 추가하면 $O(logN)$ 시간에 수행될수 있다. 임의의 remove연산은 Decreasekey연산과 deleteMin연산을 결합하여 $O(logN)$시간에 수행될수 있다.

요약

이 장에서는 우선권대기렬ADT에 대한 여러가지 실현과 리용을 고찰하였다. 표준2진더미의 실현은 그의 단순성과 속도로 하여 대단히 품위가 있다. 그것은 련결을 요구하지 않고 오직 상수적인 림시공간만을 요구하며 여전히 우선권대기렬연산들을 효과적으로 지원한다.

또한 추가적인 merge연산을 고찰하였고 그자체의 유일한 방법으로 3개의 실현들을 개발하였다. 왼쪽더미는 재귀의 위력을 보여 주는 아주 좋은 실례이다. 경사더미는 평형기준의 부족으로 하여 보기 드문 자료구조를 표현한다. 그에 대한 분석은 제11장에서 고찰하게 된다. 2항대기렬은 하나의 간단한 구상이 좋은 시간한계를 달성하는데 어떻게 리용될수 있는가를 보여 준다.

또한 조작체계들의 일정계획으로부터 모의에 이르기까지의 범위에서 우선권대기렬에 대한 여러가지 리용을 보았다. 제7장과 제9장, 제10장에서 그의 리용을 다시 고찰한다.

련습문제

6-1. insert와 findMin연산들은 둘 다 상수시간에 실행될수 있는가.

6-2. ㄱ. 초기의 빈 2진더미에 10, 12, 1, 14, 6, 5, 8, 15, 3, 9, 7, 4, 11, 13, 2를 한번에 하나씩 삽입한 결과를 표시하시오.

ㄴ. 우와 같은 입력을 리용하여 2진더미를 구축하는 선형시간알고리듬을 리용한 결과를 표시하시오.

6-3. 앞의 련습문제의 더미에서 3개의 deleteMin연산들을 실행한 결과를 표시하시오.

6-4. N개의 요소에 대한 완전2진나무는 배렬의 위치들을 1~N까지 리용한다. 불완

전한 2진나무의 배렬표현을 리용하려고 한다. 다음의 조건을 만족시키기 위하여서는 배렬이 얼마나 커야 하는가를 결정하시오.

ㄱ. 두개의 여분준위들을 가지는 2진나무(어느 정도 불균형적인).

ㄴ. 2og$N$깊이에서 최대의 깊은 매듭들을 가지는 2진나무

ㄷ. 4.1log$N$깊이에서 최대깊이의 매듭들을 가지는 2진나무

ㄹ. 최악의 경우의 2진나무

6-5. neginf감시매듭을 리용하여BinaryHwap클라스를 다시 작성하시오.

6-6. 그림 6-10에 있는 큰 더미에는 몇개의 매듭들이 있는가.

6-7. ㄱ. 2진더미에서 buildHeap연산은 요소들사이에 기껏해서 $2N-2$번의 비교를 진행한다는것을 증명하시오.

ㄴ. 8개 요소를 가지는 더미는 더미요소들사이에서 8번의 비교로 만들어 진다는것을 보여 주시오.

**ㄷ. $\frac{13}{8}N+O(\log N)$번의 요소비교로 2진더미를 구축하는 알고리듬을 작성하시오.

6-8. 더미에 있는 최대값항목에 대하여 다음과 같은것을 증명하시오.

ㄱ. 최대값은 잎매듭들중의 하나에 있어야 한다.

ㄴ. 정확히 $\lceil N/2 \rceil$개의 잎매듭들이 있다.

ㄷ. 최대값을 찾기 위해서는 모든 잎매듭들을 검사하여야 한다.

****6-9.** $N=2^k-1$인 큰 완전더미에서 k번째 제일 작은 요소의 기대되는 깊이는 log$k$로 제한된다는것을 증명하시오.

6-10. *ㄱ. 2진더미에서 어떤 값 X보다 작은 모든 매듭들을 찾기 위한 알고리듬을 작성하시오. 알고리듬은 $O(k)$로 실행될것이며 여기서 k는 출력하는 매듭들의 개수이다.

ㄴ. 이 알고리듬을 이 장에서 설명된 임의의 다른 더미구조로 확장할수 있는가.

*ㄷ. 기껏해서 대략 $3N/4$번의 비교를 진행하여 2진더미에 있는 임의의 항목 X를 찾는 알고리듬을 작성하시오.

****6-11.** N개의 요소를 가지는 2진더미에서 $O(M+\log N \log\log N)$시간에 M개의 매듭을 삽입하는 알고리듬을 작성하시오. 그리고 시간한계를 증명하시오.

6-12. N개요소들을 가지고 다음과 같은것을 수행하는 프로그람을 작성하시오.

ㄱ.N개의요소들을 더미에 하나씩 삽입하시오.

ㄴ. 더미를 선형시간에 구축하시오.

그리고 우연적인 입력정렬 및 역정렬에 대하여 2개의 알고리듬의 실행시간을 비교하시오.

6-13. 매 deleteMin연산은 최악의 경우에 $2\log N$번의 비교를 수행한다.

*ㄱ. deleteMin연산이 요소들사이에서 오직 $\log N+\log\log N+O(1)$번의 비교를 수행하도록 알고리듬을 작성하시오. 이 연산에서는 자료를 이동할 필요가 없다.

**ㄴ. $\log N+\log\log N+O(1)$의 비교가 실행되도록 ㄱ에 대한 알고리듬을 확장하시오.

**ㄷ. 이 방법을 어느 정도 리해할수 있는가?

ㄹ. 비교들을 감소시키면 알고리듬의 복잡성증가로 인한 손실이 보상되는가?

6-14. d-더미가 배렬로 기억된다면 i에 할당된 입력점에 대한 부모들과 자식들은 어디에 있는가?

6-15. 초기에 N개의 요소를 가지는 d-더미에 대하여 M개의 percolateUp연산과 N개의 deleteMin연산을 실행하여야 한다고 가정하자.

ㄱ. M, N, d에 모든 연산들의 총체적인 실행시간은 얼마인가?

ㄴ. $d=2$이면 모든 더미연산들의 실행시간은 얼마인가?

ㄷ. $d=\Theta(N)$이면 총 실행시간은 얼마인가?

*ㄹ. d를 어떻게 선택하면 총 실행시간이 최소화되는가?

6-16. 명시적인 련결을 리용하여 2진더미를 표현한다고 하자. 암시적인 위치 i에 있는 나무매듭을 찾기 위한 간단한 알고리듬을 작성하시오.

6-17. 2진더미가 명시적인 련결을 리용하여 표현된다고 하자. 여기에 lhs 와 rhs를 병합하는 2진더미에 대한 문제가 있다. 두더미가 다 각각 2^l-1과 2^r-1개의 매듭들을 포함하는 완전나무들이라고 가정하자.

ㄱ. $l=r$인 때 2개의 더미를 병합하기 위한 $O(logN)$알고리듬을 작성하시오.

ㄴ. $|l-r|=1$인 때 2개의 더미를 병합하기 위한 $O(logN)$알고리듬을 작성하시오.

*ㄷ. l과 r에는 무관계하게 2개의 더미를 병합하기 위한 $O(\log^2 N)$인 알고리듬을 작성하시오.

6-18. 최소-최대더미는 deleteMin과 deleteMax둘다를 매 연산당 $O(logN)$동안에 실현하는 자료구조이다. 그 구조는 2진더미와 동일하지만 그 더미순서속성은 짝수깊이에 있는 임의의 매듭 X에 대하여 X에 보관된 요소는 부모보다 더 작지만 조부모보다는 더 크다(그러한 경우가 있다.). 그리고 홀수깊이에 있는 임의의 매듭 X에 대해서도 X에 보관된 요소는 부모보다는 크지만 조부모보다는 더 작다(그림 6-44).

ㄱ. 최대값과 최소값요소들을 어떻게 찾는가?

*ㄴ. 최소-최대더미에 새로운 매듭을 삽입하는 알고리듬을 작성하시오.

*ㄷ. DeleteMin과 deleteMax를 수행하는 알고리듬을 작성하시오.

*ㄹ. 최소-최대더미를 선형시간에 구축할수 있는가?

**ㅁ. DeleteMin과 deleteMax, merge를 지원하여야 한다고 하자. 이때 $O(logN)$시간에 모든 연산을 지원하는 자료구조를 제기하시오.

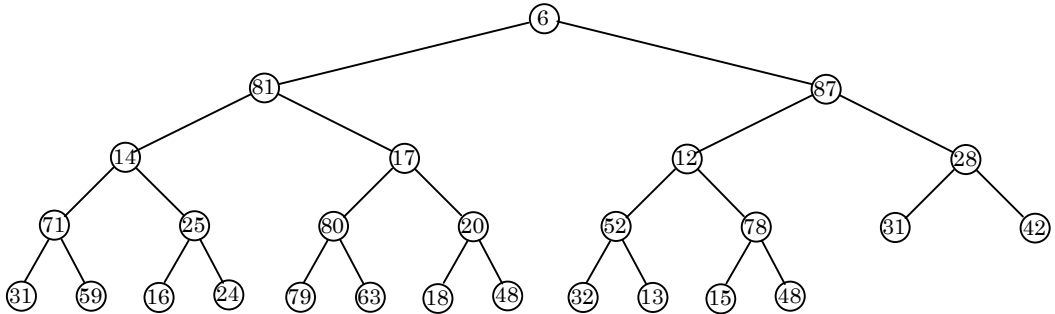



그림 6-44. 최소-최대더미

6-19. 그림 6-45에 있는 두개의 왼쪽더미들을 병합하시오.

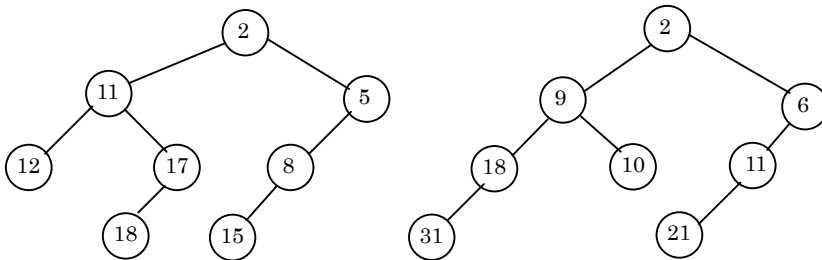



그림 6-45. 련습문제 6-19와 6-26을 위한 입력

6-20. 초기에는 비여 있는 왼쪽더미에 차례로 열쇠 1부터 15까지를 삽입한 결과를 보여 주시오.

6-21. 1부터 2^{k-1}까지의 열쇠들이 차례로 초기의 빈 왼쪽더미에 삽입된다면 완전평형나무가 형성된다는것을 증명하시오.

6-22. 제일 좋은 왼쪽더미를 만드는 입력에 대한 실례를 드시오.

6-23. ㄱ. 왼쪽더미들은 decreaseKey연산을 능률적으로 지원할수 있는가.

ㄴ. 가능하다면 이것을 수행하기 위해서 어떤 변화들이 요구되는가.

6-24. 임의의 왼쪽더미에서 지적된 위치에 있는 매듭들을 삭제하는 한가지 방법은 지연삭제를 리용하는것이다. 매듭을 삭제할 때 그것이 삭제되였다고 명백히 표시하시오. findMin이나 deleteMin을 실행할 때 만약 그 뿌리에 삭제표시가 되여 있다면 거기에는 어떤 문제가 있다. 그것은 그때 뿌리매듭은 실제적으

로 지워 져야 하고 또한 실지 최소값을 찾아야 하기때문이다. 이것은 다른 표식 붙은 매듭들을 삭제할 때에도 발생하게 된다. 이 방법에서 remove연산들은 하나의 비용단위를 가지지만 deleteMin연산이나 findMin연산의 비용은 삭제된 표식이 붙은 매듭들의 개수에 관계된다. DeleteMin연산이나 findMin 연산이후에는 연산을 실행하기전보다 k개 더 적은 표식 붙은 매듭들이 있다고 가정하자.

*ㄱ. deleteMin연산을 $O(k\log N)$시간에 실행하기 위한 방법을 보여주시오.

**ㄴ. deleteMin연산을 $O(k\log(2N/k))$시간에 실행하는 실현방법을 제기하시오.

6-25. 매 요소들을 한개 매듭의 왼쪽더미로 보고 이러한 모든 더미들을 대기렬에 넣고 다음의 단계들을 수행하면 왼쪽더미에 대하여 buildHeap를 선형시간내에 실행할수 있다. 즉 한개의 더미만 대기렬에 남을 때까지 두개의 선두요소를 꺼내여 병합하여 결과를 대기렬에 넣는다.

ㄱ. 이 알고리듬이 최악의 경우에 $O(N)$이라는것을 증명하시오.

ㄴ. 이 알고리듬이 왜 책에서 서술된 알고리듬보다 더 좋다고 생각하는가

6-26. 그림 6-45에 있는 두개의 경사더미들을 병합하시오.

6-27. 경사더미에 1~15까지의 열쇠를 차례로 삽입한 결과를 보여 주시오.

6-28. $1\sim2^k-1$까지의 열쇠를 초기의 빈 경사더미에 차례로 삽입한다면 완전평형나무가 된다는것을 증명하시오.

6-29. N개 요소들의 경사더미는 표준2진더미알고리듬을 리용하여 만들수 있다. 경사더미들에 대하여 $O(N)$실행시간을 얻는데 련습문제 6-25에서 서술한것과 같은 병합방법을 리용할수 있는가.

6-30. 2항나무 B_k가 뿌리의 자식들로서 $B_0, B_1, \ldots, B_{k-1}$을 가진다는것을 증명하시오.

6-31. 높이가 k인 2항나무가 깊이 d에 $\binom{k}{d}$개의 매듭들을 가진다는것을 증명하시오.

6-32. 그림 6-46에 있는 두개의 2항대기렬들을 병합하시오.

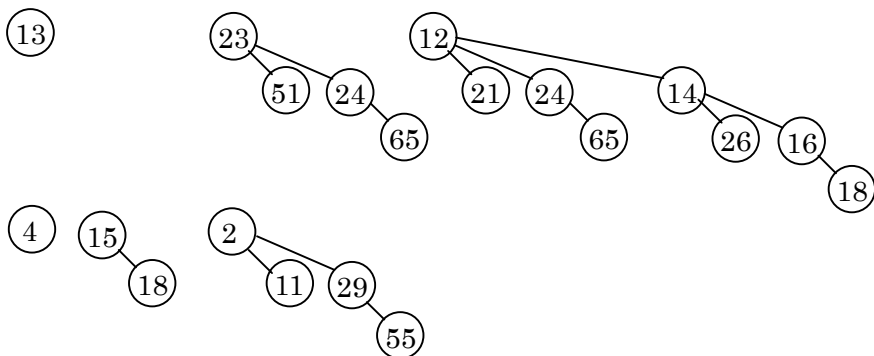



그림 6-46. 련습문제 6-32를 위한 입력

6-33. ㄱ. 초기에는 빈 2항대기렬에 대한 N개의 insert연산들은 최악의 경우에 $O(N)$시간을 가진다는것을 증명시오.

ㄴ. 요소들사이에 많아서 N-1번의 비교를 리용하여 N개 요소들을 가지는 2항대기렬을 구축하기 위한 알고리듬을 작성하시오.

*ㄷ. 최악의 경우 $O(M+\log N)$시간에 N개 요소루틴 2항대기렬에 M개 매듭을 삽입하는 알고리듬들을 작성하시오. 그리고 그 한계를 증명하시오.

6-34. 2항대기렬을 리용하여 insert연산을 수행하는 능률적인 루틴을 작성하시오. merge는 호출하지 마시오.

6-35. 2항대기렬에 대하여

ㄱ. H_1에는 어떤 나무도 남아 있는것이 없고 carry나무가 NULL이면 병합을 끝내도록 merge루틴을 수정하시오.

ㄴ. 더 작은 나무가 언제나 더 큰 나무에 병합되도록 merge를 수정하시오.

**6-36. 구조당 같은 높이를 가지는 나무를 기껏해서 2개 할당하도록 2항대기렬을 확장한다고 하자. 다른 연산들에 대하여 $O(logN)$시간을 보존하는 동안 삽입에 대하여 최악의 경우 $O(logN)$시간을 얻을수 있는가?

6-37. 매 통에는 총체적으로 무게 C를 넣을수 있고 $i_1, i_2, \cdots, i_N$항목들에 대하여 각각 $w_1, w_2, \cdots, w_N$까지의 무게를 가지는 통의 번호가 있다고 하자. 이 객체는 임의의 통에는 그의 용량보다 더 많은 무게를 넣지 않으며 가능한껏 적은수의 통을 리용하여 모든 항목을 넣는것이다. 실례로 C=5이고 항목들이 무게 2, 2, 3, 3을 가진다면 두개의 통을 가지고 이 문제를 풀수 있다. 일반적으로 이 문제는 매우 어려우며 효과적인 풀기방법도 없다. 다음과 같은 근사적인 방법들을 효과적으로 실행하기 위한 프로그람들을 작성하시오.

*ㄱ. 첫번째 그 통에 적합한 무게를 붙이시오. 충분한 공간을 가진 통이 없으면 통을 새로 만드시오. (이 전략과 이에 따르는 모든것들은 부분적으로 최적인 3개의 통을 준다.)

ㄴ. 제일 큰 공간을 가진 통에 무게를 붙이시오.

*ㄷ. 항목들을 넘쳐 나지 않게 넣을수 있는 제일 가장 충만된 통에 무게를 붙이시오.

**ㄹ. 항목들을 그의 무게를 가지고 미리 정렬하면 이 전략을 개선할수 있는가?

6-38. decreaseAllKeys(△)연산을 더미클라스에 추가하려고 한다. 이 연산의 결과는 더미내의 모든 열쇠들을 △량만큼 감소시키는것이다. 선택한 더미실현에 대하여 모든 다른 연산들은 원래의 실행시간을 유지하고 decreaseAllKey연산은

$O(1)$에 실행하도록 하는데 필요한 수정부분들을 설명하시오.

6-39. 두개의 선택알고리듬가운데서 어느것이 더 좋은 시간한계를 가지는가?

6-40. 왼쪽더미에 대하여 표준적인 operater=와 makeEmpty연산들은 재귀적호출이 너무 많기때문에 실패할수 있다. 비록 이것은 2진탐색나무들에 대해서는 옳은것이라고 하더라도 왼쪽더미에서는 더 많은 문제성이 제기된다. 왜냐하면 왼쪽더미는 매우 깊기때문이다. 한편 그것은 기본연산들에 대하여 최악의 경우 완전한 실현을 가진다. 그러므로 operator=와 makeEmpty연산들은 왼쪽더미에서 깊은 재귀를 피하기 위해 다시 실현하여야 한다.

ㄱ. t -> left에 대한 재귀호출을 t -> right에 대한 재귀호출로 되도록 재귀루틴들을 다시 작성하시오.

ㄴ. 마지막명령문이 왼쪽 부분나무에 대한 재귀호출이 되도록 루틴들을 다시 작성하시오.

ㄷ. 꼬리재귀를 제거하시오.

ㄹ. 이 함수들은 여전히 재귀이다. 남아 있는 재귀의 깊이에 대한 정확한 한계를 주시오.

*ㅁ. 경사더미에서 operator=와 makeEmpty를 다시 작성하는 방법을 설명하시오.

6-41. BinomialQueue 복사구축자를 실현하시오.

참고문헌

2진더미는 [28]에서 처음으로 설명하였다. 그 구축에 대한 선형시간알고리듬은 [14]에 있다.

d-더미는 [19]에서 처음으로 설명하였다. 4-더미가 어떤 환경에서는 2진더미들을 개선한다는것이 최근 결과들의 추측이다([22]). 왼쪽더미는 Crane [11]에 의하여 발명되였고 knuth[21]에서 설명하였다. 경사더미는 Sleator와 Tarjan[24]에 의해서 개발되였다. 2항대기렬은 Vuillemin[27]에 의해서 발명되였다. Brown은 상세한 분석과 실험적인 연구결과들을 제공하여 그것들이 만약 정확히 작성된다면 실천적으로 잘 실행된다는것을 보여 주었다([4]).

련습문제 6-7 ㄴ, ㄷ은 [17]에서 제기되였다. 련습문제 6-10 ㄷ는 [6]에서 제기된것이다.

평균적으로 대략 1.52*N*번의 비교를 리용하여 2진더미들을 구축하는 방법은 [23]에서 설명하였다. 왼쪽더미에서의 지연삭제는 [10]에서 제기되였다. 련습문제 6-36에 대한 해답은 [8]에서 찾을수 있다.

최소-최대더미(련습문제 6-18)들은 초기에 [1]에서 설명하였다. 그 연산의 좀 더 효과적인 실현은 [18]과 [15]에 주었다. 쌍방향우선권대기렬에 대한 또 다른 표현은 deap

와 diamond dequeue이다. 상세한 내용은 [5], [7], [9]에서 찾아 볼수 있다. 련습문제 6-18 ㅁ에 대한 풀이는 [12]와 [20]에서 주었다.

리론적으로 흥미 있는 우선권대기렬표현은 피보나치더미(Fibonacci heap)[16]인데 제11장에서 설명한다. 피보나치더미는 $O(logN)$시간을 가지는 삭제를 제외하고 모든 연산들을 $O(1)$의 유도시간에 수행한다. Relaxed heap[13]들은 최악의 경우에 동일한 한계를 가진다(merge는 제외하고). [3]의 수속은 모든 연산들에 대하여 최악의 경우 최적인 한계들을 준다. 또 다른 흥미 있는 실행은 쌍더미([15])인데 그것은 제12장에서 고찰하게 된다. 마지막으로 자료가 작은 옹근수들로 구성될 때 처리하는 우선권대기렬들은 [2]와 [26]에서 설명하였다.

1. M. D. Atkinson, J. R. Sack, N. Santoro, and T. Strothotte, "Min-Max Heaps and Generalized Priority Queues," *Communications of the ACM,* 29 (1986), 996-1000.
2. J. D. Bright, "Range Restricted Mergeable Priority Queues," *Information Processing Letters,* 47 (1993), 159-164.
3. G. S. Brodal, "Worst-Case Efficient Priority Queues," *Proceedings of the Seventh Annual ACM-SIAM Symposium on Discrete Algorithms* (1996), 52-58.
4. M. R. Brown, "Implementation and Analysis of Binomial Queue Algorithms," *SIAM Journal on Computing, 7* (1978), 298-319.
5. S. Carlsson, "The Deap—A Double-Ended Heap to Implement Double-Ended Priority Queues," *Information Processing Letters,* 26 (1987), 33-36.
6. S. Carlsson and J.Chen, "The Complexity of Heaps," *Proceedings of the Third Symposium on Discrete Algorithms* (1992), 393-402.
7. S. Carlsson, J. Chen, and T. Strothotte, "A Note on the Construction of the Data Structure 'Deap'," *Information Processing Letters,* 31 (1989), 315-317.
8. S. Carlsson, J. I. Munro, and P. V. Poblete, "An Implicit Binomial Queue with Constant Insertion Time," *Proceedings of First Scandinavian Workshop on Algorithm Theory* (1988), 1-13.
9. S. C, Chang and M. W. Due, "Diamond Deque: A Simple Data Structure for Priority Deques," *Information Processing Letters,* 46 (1993), 231-237.
10. D. Cheriton and R. E. Tarjan, "Finding Minimum Spanning Trees," *SIAM Journal on Computing, 5* (1976), 724-742.
11. C. A. Crane, "Linear Lists and Priority Queues as Balanced Binary Trees," *Technical Report STAN-CS-72-259,* Computer Science Department, Stanford University, Stanford, Calif., 1972.
12. Y. Ding and M. A. Weiss, "The Relaxed Min-Max Heap: A Mergeable Double-Ended

Priority Queue," *Acta Informatica, 30* (1993), 215-231.

13. J. R. Driscoll, H. N. Gabow, R. Shrairman, and R. E. Tarjan, "Relaxed Heaps: An Alter-native to Fibonacci Heaps with Applications to Parallel Computation," *Communications of theACM,* 31 (1988), 1343-1354.
14. R. W. Floyd, "Algorithm 245: Treesort 3," *Communications of the ACM, 7* (1964), 701.
15. M. L. Fredman, R. Sedgewick, D. D. Sleator, and R. E. Tarjan, "The Pairing Heap: A New Form of Self-adjusting Heap," *Algonthmica,* 1(1986), 111-129.
16. M. L. Fredman and R. E. Tarjan, "Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms," *Journal of the ACM,* 34 (1987), 596-615.
17. G. H. Gonnet and J. I. Munro, "Heaps on Heaps," *SIAM Journal on Computing, 15* (1986), 964-971.
18. Hasham and J. R. Sack, "Bounds for Min-max Heaps," *BIT,* 27 (1987), 315-323.
19. D. B. Johnson, "Priority Queues with Update and Finding Minimum Spanning Trees," *Information Processing Letters,* 4 (1975), 53-57.
20. C. M. Khoong and H. W. Leong, "Double-Ended Binomial Queues," *Proceedings of the Fourth Annual International Symposium on Algorithms and Computation* (1993), 128-137.
21. D. E. Knuth, *The Art of Computer Programming, Vol. 3: Sorting and Searching,* 2d ed., Addison-Wesley, Reading, Mass., 1998.
22. LaMarca and R. E. Ladner, "The Influence of Caches on the Performance of Sorting," *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms* (1997), 370-379.
23. C. J. H. McDiarmid and B. A. Reed, "Building Heaps Fast," *Journal of Algorithms,* 10 (1989), 352-365.
24. D. D. Sleator and R. E. Tarjan, "Self-adjusting Heaps," *SIAM Journal on Computing,* 15 (1986), 52-69.
25. T. Strothotte, P. Eriksson, and S. Vallner, "A Note on Constructing Min-max Heaps," *BIT,* 29 (1989), 251-256.
26. P. van Emde Boas, R. Kaas, and E. Zijlstra, "Design and Implementation of an Efficient Priority Queue," *Mathematical Systems Theory,* 10 (1977), 99-127.
27. J. Vuillemin, "A Data Structure for Manipulating Priority Queues," *Communications of the ACM,* 21 (1978), 309-314.
28. J. W. J. Williams, "Algorithm 232: Heapsort," *Communications of the ACM, 7* (1964), 347-348.

제7장. 정렬

이 장에서는 요소들의 배렬을 정렬하는 문제를 설명한다. 코드가 더 일반적인 대상들에 리용되지만 실례들에서 취급하는 배렬은 오직 옹근수들만 포함한다고 가정한다. 이 장의 많은 부분에서는 또한 전체 정렬이 주기억에서 수행될수 있다고 가정한다. 따라서 요소들의 개수는 상대적으로 작게 취한다(100만보다 작게). 주기억에서는 수행할수 없고 디스크나 테프상에서 수행되여야 하는 정렬도 아주 중요하다. 외부정렬이라고 하는 이런 형태의 정렬은 이 장의 마지막에서 설명한다.

내부정렬에 대한 연구는 다음과 같은것을 보여 준다.

- 삽입정렬과 같이 계산량이 $O(N^2)$인 여러개의 간단한 정렬알고리듬들이 있다.
- 코드화가 매우 단순한 쉘정렬과 같은 알고리듬도 있다. 그것은 계산량이 $O(N^2)$으로 실행되며 실천적으로도 능률적이다.
- 계산량이 $O(N\log N)$이면서 약간 더 복잡한 정렬알고리듬들이 있다.
- 임의의 일반적인 목적에 리용되는 정렬알고리듬들은 $\Omega(N\log N)$번의 비교를 요구한다.

이 장의 마지막에서는 여러가지 정렬알고리듬들을 선택하고 분석한다. 이 알고리듬들에는 알고리듬설계와 마찬가지로 코드최량화를 위한 흥미 있으면서도 중요한 방법을 포함되여 있다. 정렬은 또한 분석을 정확히 진행할수 있는 실례이기도 하다. 어떤 정렬알고리듬이 어디에 적당한가를 미리 알아야 하므로 가능한한 많은 분석을 진행해야 한다.

제1절. 예비적인 준비

여기에서 서술하게 되는 알고리듬들은 모두 서로 교환될수 있는것들이다. 매 알고리듬들에는 요소들을 포함하는 어떠한 배렬로 넘겨 지는데 모든 배렬위치들에 정렬되여야 할 자료를 가지고 있다고 가정한다. N은 정렬루틴을 통과해야 할 요소들의 개수라고 가정한다.

또한 < 와 >연산자들도 있다고 가정한다. 그 연산자들은 모순이 없는 순서로 입력을 진행하게 하는데 리용할수 있다. 이것들은 대입연산자이외에 입력자료에 대하여 할당되는 연산들이다. 이러한 조건에서의 정렬을 **비교에 기초한 정렬**(*comparison-based sorting*)이라고 한다.

제2절. 삽입정렬

1. 알고리듬

가장 단순한 정렬알고리듬의 하나가 삽입정렬이다. 삽입정렬은 N-1번의 단계로 이루어 진다. 삽입정렬은 P=1부터 N-1까지의 단계에서 0부터 P까지의 요소는 정렬된 순서로 되여 있다는 사실을 리용한다. 삽입정렬은 $0 \sim p-1$까지의 위치에 있는 요소들은 이미 정렬되여 있다는 사실을 리용한다. 표 7-1은 배렬정렬의 매 단계에 대한 배렬상태의 실례를 보여 준다.

표 7-1. 매 단계이후의 삽입정렬

초기배렬	34	8	64	51	32	21	이동된 위치들
P=1단계이후	8	34	64	51	32	21	1
P=2단계이후	8	34	64	51	32	21	0
P=3단계이후	8	34	51	64	32	21	1
P=4단계이후	8	32	34	51	64	21	3
P=5단계이후	8	21	32	34	51	64	4

표 7-1은 삽입정렬에 대한 일반적인 방법을 보여 준다. 단계 p에서는 p위치에 있는 요소를 처음의 p-1개의 요소가운데서 그의 정확한 위치가 찾아 질 때까지 왼쪽으로 이동시킨다. 프로그람 7-1에 있는 코드는 이 방법을 리용한것이다. 2~5의 행들은 교환을 충분하게 리용하지 않고 자료를 이동한다. 위치 p에 있는 요소는 tmp에 보관되며 p위치앞에 있는 더 큰 요소들을 모두 오른쪽으로 이동시킨다. 그러면 tmp는 정확한 위치에 배치된다. 이것은 2진더미들의 실현에서 리용된것과 같은 수법이다.

```
            /**
             * Simple insertion sort.
             */
            template <class Comparable>
            void insertionSort( vector<Comparable> & a )
            {
                int    j;
/* 1*/          for( int p = 1; p < a.size( ); p++ )
                {
/* 2*/              Comparable tmp = a[ p ];
/* 3*/              for( j = p; j > 0 && tmp < a[ j - 1 ]; j -- )
/* 4*/                  a[ j ] = a[ j - 1 ];
```

```
/* 5*/                a[ j ] = tmp;
                    }
            }
```

프로그람 7-1. 삽입정렬루틴

2. 삽입정렬의 분석

매개 요소의 삽입이 N번 반복되는 다중순환고리들때문에 삽입정렬은 계산량이 $O(N^2)$으로 된다. 더우기 자료가 역순서로 입력되는 경우에 이 한계를 주기때문에 명백하다. 정확히 계산하면 3행에 있는 순환한계검사가 P의 매 값에 대하여 거꺼해서 p +1번 실행된다.

모든 p에 대하여 합하면 총체적으로

$$\sum_{i=2}^{N} i = 2+3+4+\cdots+N = \Theta(N^2)$$

이 나온다.

한편 입력이 이미 정렬되여 있다면 그 실행시간은 $O(N)$이다. 그것은 안쪽 순환고리에서의 검사가 항상 그 즉시에 실패하기때문이다. 사실 입력이 거의 정렬되여 있다면(이것은 다음절에서 엄밀하게 정의된다.) 삽입정렬은 고속으로 실행된다. 이처럼 변화범위가 넓기때문에 이 알고리듬의 평균적인 동작을 분석할 가치가 있다. 다음절에서 보게 되는 것처럼 다른 정렬알고리듬에서와 마찬가지로 평균적인 경우에 삽입정렬은 계산량이 $\Theta(N^2)$이다.

제3절. 간단한 정렬알고리듬의 아래한계

수들의 배렬에서 **반전**(*inversion*)이란 $i<j$이지만 a[i]>b[j]인 성질을 가지는 임의의 순서 붙은 쌍 (i ,j)을 말한다. 앞절의 실례에서 입력목록 34, 8, 64, 51, 32, 21은 9개의 반전들을 가진다. 즉 (34, 8), (34, 32), (34, 21),(64, 51), (64, 32), (64, 21),(51, 32), (51, 21), (32, 21)이다. 이것은 삽입정렬에 의해서 (암시적으로) 수행되여야 하는 교환들의 수를 의미한다. 이것은 항상 있게 되는 경우이다. 왜냐하면 순서가 바뀐 두개의 린접한 요소들을 항상 교환하면 반전이 정확히 하나씩 제거되며 따라서 정렬된 배렬에는 반전이 없게 되기때문이다. 알고리듬에 포함된 다른 처리를 계산량 $O(N)$에 포함시켜야 하므로 삽입정렬의 실행시간은 $O(I+N)$이다. 여기서 I는 처음배렬의 반전수이다. 따라서 삽입정렬은 반전수가 $O(N)$이면 선형시간에 실행된다.

순렬로 평균반전수를 계산하는 방법으로 삽입정렬의 평균실행시간에 대한 정확한 한계를 계산할수 있다. 일반적으로 평균에 대하여 정의한다는것은 어려운 일이다. 중복되는 요소는 전혀 없다고 가정한다(만약 중복을 허용한다면 중복의 평균개수가 몇개인가 하는 것은 명백하지 않다.). 이 전제를 리용하여 입력은 첫 N개의 옹근수(오직 상대적인 순서만 중요하기때문에)들의 임의의 순렬이고 그것들은 모두 거의 같다고 가정할수 있다. 이 전제밑에서 다음의 정리를 얻는다.

정리 7-1.

N개의 서로 다른 요소들을 가지는 배렬에서 평균반전수는 $N(N-1)/4$이다.

증명:

어떤 요소들의 목록 L에 대한 거꿀순서로 된 목록을 L_r라고 하자. 실례에서의 거꿀순서목록은 21, 32, 51, 64, 8, 34이다. 목록에서 $y > x$인 두 요소들의 모든 쌍 (x, y)를 고찰하자. 이 순서 붙은 쌍은 명백히 L과 L_r중의 어느 하나에서 반전을 나타낸다. 목록 L과 그의 거꿀순서목록 L_r에서 이 쌍들의 총 개수는 $N(N-1)/2$이다. 따라서 목록은 평균 이 량의 절반 즉 $N(N-1)/4$의 반전을 가진다.

이 정리는 삽입정렬이 평균인 경우 2차적이라는것을 암시한다. 이것은 또한 린접요소들에 대하여 단순히 교환만 진행하는 임의의 알고리듬에 대한 아주 엄밀한 아래한계를 준다.

정리 7-2.

린접요소들을 교환하여 정렬하는 알고리듬은 평균적으로 $\Omega(N^2)$시간을 요구한다.

증명:

반전들의 평균수는 초기에 $N(N-1)/4=\Omega(N^2)$이다. 매 교환에서 오직 하나의 반전만이 제거되므로 따라서 $\Omega(N^2)$의 교환이 요구된다.

이것은 아래한계증명에 대한 하나의 실례이다. 이것은 린접들의 교환을 암시적으로 수행하는 삽입정렬뿐아니라 거품정렬과 선택정렬(여기에서는 서술하지 않는다.)과 같은 다른 단순한 알고리듬에 대해서도 타당하다. 사실상 이것은 단지 린접들의 교환만을 수행하는 다른 정렬알고리듬들을 포함하여 정렬알고리듬 클라스의 전반에서 유효하다. 때문에 이 증명을 경험적으로는 확증할수 없다. 비록 이 아래한계증명이 단순하다고 해도 일반적인 아래한계증명은 웃한계증명보다 상당히 더 복잡하며 일부 경우에는 신기할 정도이다.

이 아래한계는 정렬알고리듬이 부분2차 즉 $O(N^2)$의 시간에서 실행되도록 하기 위하여서는 멀리 떨어 져 있는 요소들사이에서 비교 특히는 교환을 진행하여야 한다는것을 보여 준다. 정렬알고리듬은 반전들을 제거하는 방법으로 실행되며 효과적인 실행을 위해서는 교환할 때마다 하나이상의 반전들을 제거하여야 한다.

제4절. 쉘정렬

쉘정렬(Donald shell에 의해서 제안된)은(비록 부분2차시간한계가 증명된 초기의 발견이후 여러해 지난후에야 비로소 알게 된것이지만) 2차적인 시간한계로 처리되는 첫 알고리듬들중의 하나였다. 앞절에서 본것처럼 그것은 떨어 져 있는 요소들을 비교하여 처리하는데 비교하는 요소들사이의 거리는 알고리듬의 마지막단계를 실행할 때 감소한다.
그 마지막단계에서는 린접요소들이 비교된다. 이 리유로 쉘정렬은 때때로 **증분감소정렬**(*diminishing increment sort*)이라고도 한다.

쉘정렬은 **증분렬**(*increment sequence*)이라고 하는 $h_1, h_2, \cdots, h_t$의 렬을 리용한다. 임의의 증분렬은 h_1=1인 조건에서는 계속 처리되지만 일부 선택의 결과가 다른것들의 경우보다 더 좋을 때가 있다(이에 대해서는 후에 설명한다). 하나의 단계가 처리된후 어떤 증분 h_k를 리용하여 매 i에 대해 $a[i] \leq a[i+h_k]$를 주는데 h_k만큼 떨어 진 모든 요소들이 정렬된다. 이때 파일에 대한 정렬을 h_k정렬이라고 한다. 실례로 표 7-2는 쉘정렬의 여러개 단계들에 대한 배렬상태를 보여 준다. 쉘정렬의 중요한 특성을 증명없이 설명하면 그다음 h_{k-1}-정렬된 임의의 h_k-정렬파일이 h_k-정렬되여 남아 있다는것이다. 만약 그렇지 않다면 이 알고리듬은(앞의 단계들에서 수행된 처리가 다음의 단계들에서 수행되지 않으므로) 효과가 적어 질것이다.

표 7-2. 쉘정렬의 매 단계

초기배렬	**81**	**94**	**11**	**96**	**12**	**35**	**17**	**95**	**28**	**58**	**41**	**75**	**15**
5-정렬한 다음	35	17	11	28	12	41	75	15	96	58	81	94	95
3-정렬한 다음	28	12	11	35	15	41	58	17	94	75	81	96	95
1-정렬한 다음	11	12	15	17	28	35	41	58	75	81	94	95	96

h_k정렬하는 일반적인 방법은 매 위치 i에 대하여 h_k, h_{k+1}, $\cdots$, N-1에서 요소를 i, $i-h_k$, $i-2h_k$ $\cdots$ 가운데의 정확한 위치에 배치한다. 비록 이것은 그 실현에 영향을 미치지 않는다 해도 구체적으로 조사해 보면 h_k-정렬이 h_k개의 독립적인 부분배렬들에 대한 삽입정렬이라는것을 보여 준다. 이 고찰은 쉘정렬의 실행시간을 분석할 때 중요하다.

증분렬에 대한 일반적인 선택(그러나 충분하지 못한)은 쉘에 의해서 제시된 증분렬

즉 $h_t = \lfloor N/2 \rfloor$, $h_k = \lfloor h_{k+1}/2 \rfloor$을 리용하는것이다. 프로그람 7-2는 이 증분렬을 리용하여 쉘정렬을 실현하는 함수를 보여 준다. 앞으로 알고리듬의 실행시간에 의의 있는 개선을 주는 증분렬에 대하여 보게 되는데 그 약간한 변화는 알고리듬의 실현에 대단히 큰 영향을 미칠수 있다(련습문제 7-10).

프로그람 7-2의 코드에서는 삽입정렬의 실현에서와 같이 공개적으로 교환을 리용하지 않게 작성된것이다.

```
            /**
            * shellsort, using shell's (poor) increments.
            */
            template <class Comparable>
            id shellsort( vector<Comparable> & a )
            {
               int   j;
/* 1*/             for ( int gap = a.size( ) / 2; gap > 0; gap /= 2 )
/* 2*/             for( int i = gap; i < a.size( ); 1++ )
                   {
/* 3*/                  Comparable tmp = a[ i ];
/* 4*/                  for( j = i; j >= gap && tmp < a[ j - gap ]; j -= gap )
/* 5*/                       a[ j ] = a[ j - gap ];
/* 6*/                  a[ j ] = tmp;
                   }
               }
```

프로그람 7-2. 쉘의 증분렬을 리용하는 쉘정렬루틴
(더 좋은 증분들이 가능하다.)

1. 쉘정렬의 최악의 경우에 대한 분석

쉘정렬은 비록 코드화가 단순하지만 그 실행시간에 대한 분석은 전혀 다르다. 쉘정렬의 실행시간은 증분렬의 선택에 의해 정해 지므로 그에 대한 증명도 있어야 한다. 쉘정렬의 평균경우에 대한 분석은(보통 많이 쓰이는 증분렬들은 제외하고) 여러해동안 계속 진행되여 왔다. 두개의 독특한 증분렬에 대하여 엄밀한 최악의 경우의 한계들을 증명해 보자.

정리 7-3.

쉘의 증분을 리용한 쉘정렬의 최악의 경우의 실행시간은 $\Theta(N^2)$이다.

증명:

증명에서는 최악의 경우의 실행시간에 대한 웃한계뿐만아니라 실제 실행에서 $\Theta(N^2)$시간을 취하는 어떤 입력이 존재한다는것을 보여 주어야 한다. 먼저 나쁜 경우를 가지고 아래한계를 증명하자. 먼저 N을 2의 제곱이 되게 택한다. 이렇게 하면 값이 1인 마지막증분을 제외하고는 모든 증분들이 짝수가 된다. 이제 짝수위치들에는 $N/2$개의 제일 큰 수들을 가진 배렬을 그리고 홀수위치들에는 $N/2$개의 제일 작은 수들을 가진 배렬을 입력하자(이 증명에서 첫번째 위치는 위치 1이다.). 제일 마지막의것을 내놓고는 모든 증분들이 짝수이므로 마지막단계에 도달했을 때 $N/2$개의 제일 큰 수들은 여전히 모두 짝수위치에 있고 $N/2$개의 제일 작은 수들은 여전히 홀수위치에 있다. i번째로 제일 작은 수($i \leq N/2$)는 마지막단계를 시작하기전의 위치 $2i$-1에 있다. i번째 요소를 그의 정확한 위치로 회복하자면 배렬에서 그것을 i-1에로 이동하여야 한다. 그러므로 $N/2$개의 제일 작은 요소들을 정확한 위치에 배치하자면 $\sum_{i=1}^{N/2} i-1=\Omega(N^2)$번의 처리를 진행해야 한다. 실례로 표 7-3은 N=16일 때의 좋지 못한(최악의 경우는 아니다) 입력을 보여 준다. 이 정렬을 수행한 다음에 남아있는 반전의 수는 정확히 1+2+3+4+5+6+7=28이다. 그러므로 제일 마지막단계는 상당한 시간이 걸린다.

증명을 끝내기 위하여 $O(N^2)$의 웃한계를 보여 주자. 이미 고찰한것처럼 증분 h_k를 가진 단계는 약 N/h_k개의 요소들에 대한 h_k의 삽입정렬들로 이루어 진다. 삽입정렬이 2차적이라는데로부터 하나의 단계의 총체적인 비용은 $O(h_k(N/h_k)^2))=O(N^2/h_k)$이다. 모든 단계들의 합계로부터 얻어 지는 총 한계는

$$O(\sum_{i=1}^{t} N^2/h_i) = O(N^2 \sum_{i=1}^{t} 1/h_i)$$

으로 된다. 증분들은 공비가 2인 등비수렬을 형성하며 그 렬의 제일 큰 항목은 h_1=1이기때문에 $\sum_{i=1}^{t} 1/h_i < 2$이다. 따라서 이 전체 시간한계는 $O(N^2)$로 된다.

표 7-3. 쉘증분에 의한 쉘정렬의 좋지 못한 경우

초기배렬	1	9	2	10	3	11	4	12	5	13	6	14	7	15	8	16
8-정렬한다음	1	9	2	10	3	11	4	12	5	13	6	14	7	15	8	16
4-정렬한다음	1	9	2	10	3	11	4	12	5	13	6	14	7	15	8	16
2-정렬한다음	1	9	2	10	3	11	4	12	5	13	6	14	7	15	8	16
1-정렬한다음	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

쉘증분들에 대한 문제는 증분쌍들이 서로 소인수가 아니라도 되며 따라서 더 작은 증분은 약간한 효과를 가질수 있다. 히바드(Hibbard)는 좀 다른 증분렬을 암시하였다. 그

증분렬은 실천적으로나 리론적으로 더 좋은 결과들을 준다. 그의 증분들은 1, 3, 7, ..., 2^k-1과 같은 형태이다. 비록 이 증분들이 거의 같다고 해도 기본적인 차이는 련속적인 증분들이 그 어떤 공통인수도 가지지 않는다는것이다. 이제 이 증분렬을 놓고 쉘정렬에 대한 최악의 경우의 실행시간을 분석하자. 그 증명은 좀 복잡하다.

정리 7-4.
히바드의 증분들을 리용한 쉘정렬의 최악의 경우의 실행시간은 $\Theta(N^{3/2})$이다.

증명;
여기에서는 다만 웃한계에 대해서만 증명하고 아래한계에 대한 증명은 련습문제로 남긴다. 증명은 수론에서 잘 알려 진 결과들을 요구한다. 이 결과는 이 장의 마지막에서 고찰하게 된다.

앞에서처럼 웃한계에 대하여 매 단계에 대한 실행시간과 모든 단계들에서의 합을 계산한다. 증분 $h_k>N^{1/2}$에 대하여 앞의 정리로부터 한계 $O(N^2/ h_k)$를 리용한다. 이 한계는 비록 다른 증분들에 대해서도 성립하지만 너무 커서 리용할수 없다. 직관적으로 말하면 우리는 이 증분렬이 특수하다는 사실을 리용하여야 한다. 증명하려는 것은 위치 p에 있는 임의의 요소 $a[p]$에 대하여 그것이 h_k -정렬을 수행하는 시간이라면 $a[p]$보다 더 큰 위치 p의 왼쪽에는 적은 수의 요소들만이 있다는것이다.
입력배렬을 h_k-정렬할 때 이미 h_{k+1}과 h_{k+2}로 정렬되여 있다는것을 알수 있다. h_k정렬에 앞서 $i\leqq P$ $a[p-i]$인 위치 p와 $p-i$에 있는 요소들을 고찰하자. 만일 i가 h_{k+1} 혹은 h_{k+2}의 배수이라면 명백히 $a[p-i]< a[p]$이다. 그러나 만일 i가 h_{k+1}과 h_{k+2}의 선형결합(부가 아닌 옹근수로서)으로 표현할수 있다면 그때 $a[p-i]< a[p]$이다. 실례로 3-정렬할 때 파일은 이미 7- 및 15-로 정렬되여 있다. 52는 7과 15의 선형결합으로 표현할수 있는데 그것은 52=1*7+3*15이기때문이다. 따라서 $a[100]$은 $a[152]$보다 더 클수 없는데 그것은 $a[100]\leqq a[107]\leqq a[122]\leqq a[137]\leqq a[152]$이기때문이다.

현재 $h_{k+2}=2h_{k+1}+1$이므로 h_{k+1}과 h_{k+2}는 공통인수를 공유할수 없다. 이 경우에 적어도 $(h_{k+1}-1)(h_{k+2}-1)=8h_k^2+4h_k$만큼 큰 모든 옹근수들은 h_{k+1}과 h_{k+2}의 선형결합으로 표현될수 있다는것을 보여 줄수 있다(이 장의 마감에 서술한 참고문헌을 볼것).

이것은 제일 안쪽에 있는 for순환이 $N- h_k$위치들의 매개 요소에 대하여 기껏해서 $8h_k+4_=$ $O(h_k)$시간에 실행될수 있다는것을 보여 준다. 이것은 단계당 $O(Nh_k)$의 한계를 준다.

증분의 약 절반은 $h_k < \sqrt{N}$ 를 만족시킨다는 사실과 t가 짝수라는 가정으로부터 총체적인 실행시간은

$$O(\sum_{k=1}^{t/2} Nh_k + \sum_{k=t/2+1}^{t} N^2/h_k) = O(N\sum_{k=1}^{t/2} h_k + N^2 \sum_{k=t/2+1}^{t} 1/h_k)$$

이다.

량쪽 합들은 등비수렬이고 또한 $h_{t/2} = \Theta(\sqrt{N})$ 이기때문에 이것은 간단히

$$= O(Nh_{t/2}) + O(\frac{N^2}{h_{t/2}}) = O(N^{3/2})$$

으로 된다.

히바드의 증분들을 리용한 쉘정렬의 평균경우의 실행시간은 모의에 기초하여 $O(N^{5/4})$ 이라고 보지만 누구도 이것을 증명하지 못하였다. 프라트(Pratt)는 증분렬들의 넓은 범위에서 $\Theta(N^{3/2})$한계가 적용된다는것을 보여 주었다.

쎄지위크(Sedgewick)는 최악의 경우에 $O(N^{4/3})$실행시간을 주는 여러개의 증분렬들을 제안하고 그것을 실현할수 있었다. 이 증분렬들에 대한 평균실행시간은 $O(N^{7/6})$으로 추측된다. 경험적인 연구결과들은 이 증분렬들이 히바드의 증분렬보다 실천적으로 아주 더 좋게 실행된다는것을 보여 주었다. 이가운데서 제일 좋은 증분렬은 {1, 5, 19, 41, 109, … }인데 이 증분렬의 매 항들은 $9 \cdot 4^i - 9 \cdot 2^i + 1$이거나 $4^i - 3 \cdot 2^i + 1$의 형태이다. 이것은 이 값들을 배렬에 배치함으로써 아주 쉽게 실현된다. 이 증분렬은 어떤 증분렬이 쉘정렬의 실행시간에 의의 있는 개선을 줄수 있는 가능성이 적다고 하여도 실천적으로는 제일 잘 알려 진것이다.

일반적으로 수론과 조합에 대한 어려운 정리들을 요구하는 쉘정렬에 대하여서는 여러가지 다른 결과들도 있으며 그것은 주로 리론적으로 흥미가 있다. 쉘정렬은 아주 복잡한 분석을 가진 대단히 간단한 알고리듬에 대한 좋은 실례이다. 쉘정렬은 실천적으로 효과 있게 리용될수 있다. 코드의 단순성은 중간정도로 큰 입력에 대등한 정렬알고리듬을 선택하게 한다.

제5절. 더미정렬

제6장에서 언급된것처럼 우선권대기렬들은 $O(N\log N)$시간에 정렬하는데 리용될수 있다. 이 방법에 기초한 알고리듬을 **더미정렬**(*heapsort*)이라고 하며 지금까지 고찰한 방법들가운데서 제일 좋은 큰O실행시간을 준다. 그렇지만 실천적으로는 쎄지위크(sedgewick)의 증분렬을 리용한 쉘정렬보다는 더 느리다.

그의 기본방법은 제6장에서 본바와 같이 N개 요소들을 가지는 2진더미를 구축하는것이라는것을 상기하자. 이 단계들은 $O(N)$시간을 가진다. 그다음 N개의 deleteMin연산을 수행한다.더미에서 뽑아 내는 제일 작은 첫번째 요소들은 정렬된 순서로 된다. 이 요소들을 보조배렬에 기록한 다음 그 배렬을 도로 복사함으로써 N요소들을 정렬한다. 매

deleteMin연산은 $O(\log N)$시간을 가지므로 총 실행시간은 $O(N\log N)$이다.

이 알고리듬에서 기본문제는 여분의 배렬을 리용하는것이다. 따라서 기억기요구조건이 2배로 된다. 이것은 어떤 정황들에서는 문제로 될수 있다. 보조배렬을 초기의 배렬에 다시 복사하는데 걸리는 여분시간은 단지 $O(N)$이므로 실행시간에 크게 영향을 주지 않는다. 문제는 기억공간이다.

보조배렬의 리용을 피하기 위한 재치 있는 방법은 매개 deleteMin연산이후에 더미가 하나씩 줄어 든다는 사실을 리용하는것이다. 따라서 더미의 제일 마지막세포는 이전단계에서 지워 진 요소를 기억하는데 리용할수 있다. 실례로서 6개의 요소를 가진 더미가 있다고 하자. 첫 deleteMin연산은 a_1을 생성한다. 이제는 더미에 5개의 요소만 있으며 위치 6에 a_1을 넣을수 있다. 다음의 deleteMin연산은 a_2를 생성한다. 이제는 더미에 오직 4개의 요소만 있기때문에 a_2를 위치 5에 넣을수 있다.

이러한 방법을 리용하면 마지막 deleteMin연산후에 배렬의 요소들은 감소되는 순서로 배치되게 된다. 요소들을 전형적인 올리순서로 배치하려면 부모가 자식보다 더 큰 요소를 가지도록 순서속성을 변화시킬수 있다. 이렇게 하면 최대더미를 가지게 된다. 실현에서는 최대더미를 리용하지만 속도를 높이기 위하여 실제적인 ADT를 리용하지 않는다. 보통 모든 처리는 배렬에서 수행된다. 첫번째 걸음은 더미를 선형시간에 구축한다. 그다음 더미에서 제일 마지막요소를 첫번째 요소와 교환하고 더미크기를 감소시켜 아래로 려과시키는 방법으로 N-1번의 deleteMax연산들을 수행한다. 알고리듬이 끝날 때 요소들은 정렬된 순서로 배렬내에 배치되게 된다. 실례로 입력렬 31, 41, 59, 26, 53, 58, 97을 고찰하자. 이에 대한 결과적인 더미를 그림 7-1에 보여 준다.

그림 7-2는 첫 deleteMax연산이 처리된 다음의 더미를 보여 준다. 그림에서 보는것처럼 더미에서 제일 마지막 요소는 31이다. 97은 수법상 더는 2진더미부분이 아닌 더미배렬부분에 배치되였다. deleteMax연산을 5번 더 진행한 후에 더미에는 실제적으로 오직 하나의 요소만 있지만 더미배렬에 남아 있는 요소들은 정렬된 순서로 된다.

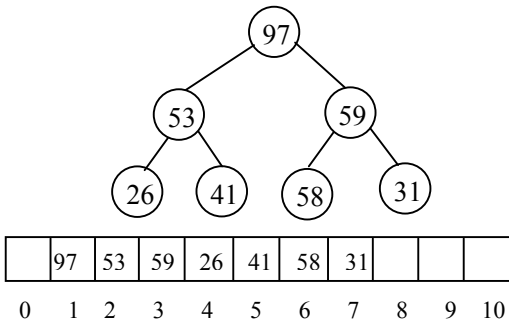



그림 7-1. biuldheap이후의 (Max)더미

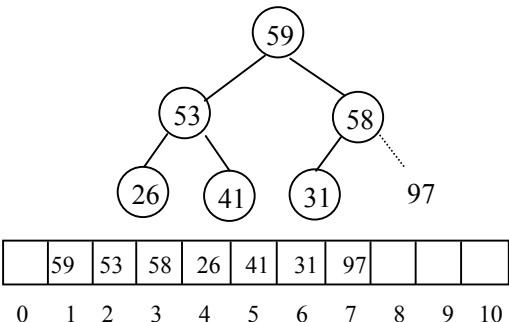



그림 7-2. 첫 deleteMax한후의 더미

더미정렬을 수행하는 코드를 프로그람 7-3에 준다. 좀 복잡한것은 자료가 배렬의 첨

수 1에서부터 배치되는 2진더미와는 달리 더미정렬을 위한 배렬에서는 위치 0에서부터 자료를 배치한다. 따라서 코드는 2진더미코드와 약간 다른데 그 변화는 적다.

```
/**
  * Internal method for heapsort.
  * i is the index of an item in the heap.
  * Returns the index of the left child.
  */
inline int leftChild( int i )
{
        return 2 * i + 1;
}

/**
* Internal method for heapsort that  is used in
* deleteMax and buildHeap.
* i is the position from which to percolate down.
* n  is the logical size of the binary heap.
*/
template <class Comparable>
void percDown( vector<Comparable> & a, int i, int n )
{
      int child;
      Comparable tmp;
/* 1*/  for( tmp = a[ i ]; leftChild( i ) < n; i = child )
       {
/* 2*/      child = leftChild( i );
/* 3*/      if( child != n - 1 && a[ child ] < a[ child + 1 ] )
/* 4*/          child++;
/* 5*/     if( tmp < a[ child ] )
/* 6*/          a[ i ] = a[ child ];
            else
/* 7*/        break;
        }
/* 8*/     a[ i ] = tmp;

}

  /**
  * Standard heapsort.
  */
  template <class Comparable>
  void heapsort( vector<Comparable> & a )
  {
/* 1*/      for( int i = a.size( ) / 2; i >= 0; 1-- ) /* buildHeap */
/* 2*/      percDown( a, i, a.size( ) );
/* 3*/      for( int j = a,size( ) - 1; j > 0; j-- )
             {
/* 4*/          swap( a[ 0 ], a[ j ] );              /* deleteMax */
```

```
/* 5*/                    percDown( a, 0, j );
                      }
                  }
```

프로그램 7-3. 더미정렬

1. 더미정렬의 분석

제6장에서 고찰한것처럼 더미를 구축하기 위한 첫번째 단계에서는 기껏해서 $2N$번의 비교를 리용한다. 두번째 단계에서 i번째 deleteMax연산은 적어도 $2\lfloor \log i \rfloor$번의 비교를 리용하며 총체적으로는 많아서 $2N\log N - O(N)$번의 비교를 리용한다($N \geq 2$)라고 가정한다.). 그 결과 최악의 경우 더미정렬에는 많아서 $2N\log N - O(N)$번의 비교가 리용된다. 련습문제 7-13은 모든 deleteMax연산들이 최악의 경우 동시에 진행될수 있다는것을 보여 준다.

경험은 더미정렬의 실현에 모순이 전혀 없다는것을 보여 준다. 평균적으로 그것은 최악의 경우의 한계들보다 약간 적은 비교를 진행한다. 그러나 현재까지 그 누구도 더미정렬의 평균실행시간에 대한 증명된 한계를 보여 주지 못하였다. 그것은 련속적인 deleteMax연산들이 확률변수들을 아주 복잡하게 만들어 더미의 우연성을 파괴하기때문이다. 최근 또 다른 방법이 성과적으로 증명되였다.

정리 7-5.

N개의 서로 구별되는 항목들의 우연순렬을 더미정렬하는데 리용되는 평균비교회수는 $2N\log N - O(N\log\log N)$이다.

증명:

더미구축단계는 평균적으로 $\Theta(N)$의 비교를 리용하며 따라서 오직 둘째 단계에서의 한계만을 증명하여야 한다. 순렬 $\{1, 2, ..., N\}$이 있다고 하자.

i번째 deleteMax연산이 뿌리요소를 d_i준위만큼 아래로 밀어 낸다고 하자. 그때 그것은 $2d_i$의 비교를 진행한다. 어떤 입력에 대한 더미정렬에서 단계2의 비용을 정의하는 하나의 비용렬 D 즉 $d_1, d_2, \cdots, d_N$이 있다. 그 비용은 $M_D = \sum_{i=1}^{N} d_i$로 주어 지며 따라서 리용되는 비교수는 $2M_D$이다.

$f(N)$을 N개 항목들을 가지는 더미들의 개수라고 하자. 그러면 $f(N) > (N/(4e))^N$라는것을 보여 줄수 있다(련습문제 7-53). 여기에서 e=2.71828…이다. 이 더미들가운데서 지수적으로 작은 분수(특히 $(N/16)^N$)들은 오직 $M = N(\log N - \log\log N - 4)$보다 더 작은 비용을 가진다는것을 보여 준다. 여기에서 M_D의 평균값은 적어도 $M - O(1)$이다. 따라서 평균비교수는 적어도 $2M$으로 된다. 여기에서 기본목적은 작은 비용렬을 가지는 아주 적

은 수의 더미들이 있다는것을 보여 주는것이다.

준위 d_i는 기껏해서 2^{d_i}개의 매듭들을 가지기때문에 뿌리요소가 임의의 d_i로 갈수 있는 가능한 위치들은 2^{d_i}에 있다. 그 결과 임의의 렬 D에 대하여 개개의 대응하는 deleteMax연산렬의 수는 기껏해서

$$S_D = 2^{d_1} 2^{d_2} \cdots 2^{d_N}$$

이다. 간단한 수학적인 조작을 하면 주어 진 렬 D에 대하여

$$S_D = 2^{M_D}$$

이라는것을 보여 준다.

매개 d_i는 1과 $\lfloor \log N \rfloor$사이의 임의의 값이라고 가정할수 있기때문에 적어도 $(\log N)^N$개의 가능한 렬 D가 있다. 정확히 M만한 비용을 요구하는 개개의 deleteMax연산렬의 수는 기껏해서 이러한 매개 비용렬들에 대하여 총체적인 비용을 가지는 비용렬들의 M배만한 수이다. $(\log N)^N 2^M$의 한계는 즉시 나온다.

M보다 작은 비용렬을 가지는 더미의 총 개수는 기껏해서

$$\sum_{i=1}^{M-1} (\log N)^N 2^i < (\log N)^N 2^M$$

이다.

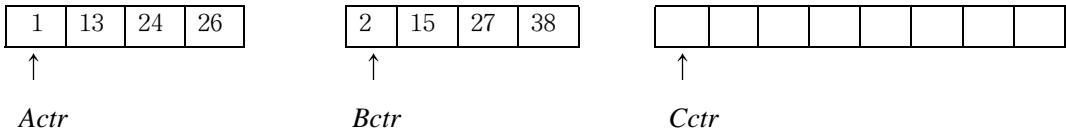
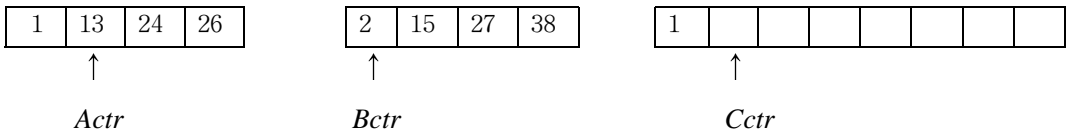
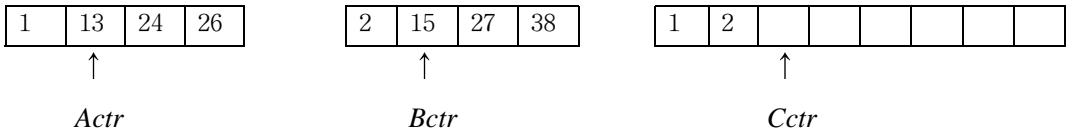
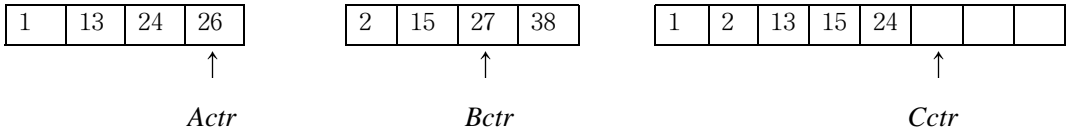
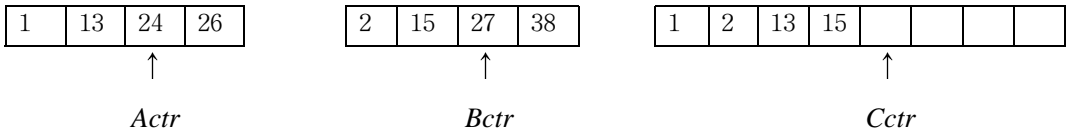
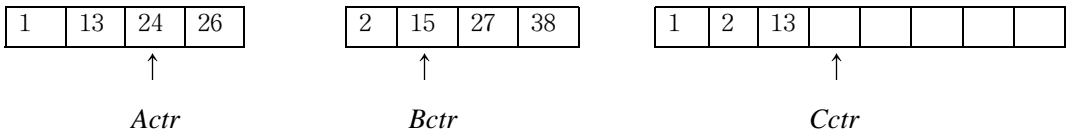
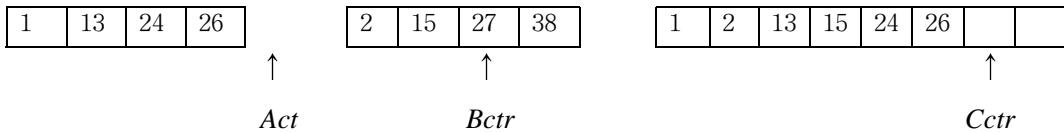
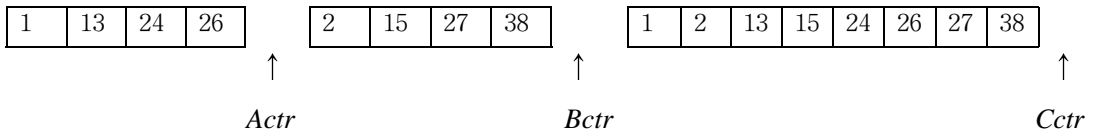
만일 $M=N(\log N-\log\log N-4)$를 선택하면 M보다 더 작은 비용렬을 가지는 더미개수는 기껏해서 $(N/16)^N$이다. 그리고 정리는 보다 앞의 주해로부터 나온다.

더 복잡한 인수를 리용하면 더미정렬은 적어도 $N\log N-O(N)$번의 비교를 늘 리용하며 이 한계를 가질수 있는 입력이 있다는것을 보여 줄수 있다. 평균경우에도 $2N\log-O(N)$번의 비교(정리 7-5의 비선형적인 두번째 항보다 더 좋다.)를 할것이라고 보는데 이에 대해서는 증명할수 있으며 공개되여 있다.

제6절. 병합정렬

이제는 병합정렬을 보자. 병합정렬은 최악의 경우에 $O(N\log N)$의 시간에 수행된다. 병합정렬에 리용되는 비교수는 거의 최량이다. 병합정렬은 재귀알고리듬의 좋은 실례이다.

이 알고리듬에서 기본연산은 두개의 정렬된 목록을 병합하는것이다. 목록이 정렬되여 있기때문에 만약 출력을 세번째 목록에 넣는다면 입력을 하나로 관통시켜 수행할수 있다. 기본병합알고리듬은 2개의 입력배렬 A와 B, 출력배렬 C 그리고 3개의 계수기 *Actr*, *Bctr*, *Cctr*를 가진다. *Actr*, *Bctr*, *Cctr*는 초기에 각각 개별적인 배렬들의 시작위치로 설정

된다. $A[Actr]$와 $B[Bctr]$중에서 더 작은것을 C의 다음 가입자에 복사한다. 그리고 대응하는 계수기들을 전진시킨다. 어느 한쪽의 입력목록이 비게 되면 다른 목록의 나머지를 C에 복사시킨다. 병합루틴이 어떻게 작업하는가를 다음의 입력에 대하여 보자.

만일 배렬 A가 1, 13, 24, 26이고 B는 2, 15, 27, 38이라면 알고리듬은 다음과 같이 수행된다. 즉 먼저 비교는 1과 2사이에서 진행된다. 1이 C에 들어 간다. 다음 13과 2를 비교하여 2를 C에 넣는다.

13과 15를 비교하여 13이 C에 들어 가며 24와 15를 비교한다.

이것을 26과 27이 비교될 때까지 진행한다.

26이 *C*에 들어 가면 배렬 *A*가 비게 된다.

배렬 *B*의 나머지가 C에 복사된다.

두개의 정렬된 목록의 병합시간은 기껏해서 N-1번의 비교가 진행되기때문에 명백히 선형이다(여기서 N은 요소의 총 개수이다.). 이것을 보려면 마지막 두개가 첨가되는 마지막비결을 제외하고는 매번의 비교가 C에 요소를 첨가하는것이라는것에 주목해야 한다.

그러므로 병합정렬알고리듬은 서술하기 쉽다. *N*=1이면 정렬하는데 오직 하나의 요소만 있으며 해답은 명백하다. *N*=1이 아닌 다른 경우에는 첫 절반과 둘째 절반을 재귀적으로 병합정렬한다. 이것은 정렬된 두개의 부분을 주는데 우에서 서술된 병합알고리듬을 리용하여 하나로 병합할수 있다. 실례로 8개 요소들의 배렬 24, 13, 26, 1, 2, 27, 38, 15를 정렬하기 위해서 처음 4개와 마지막 4개 요소들을 재귀적으로 정렬한다. 이때 1, 13, 24, 26, 2, 15, 27, 38이 얻어 진다. 다음 우에서처럼 두개의 부분으로 갈라 진것들을 병합한다. 결과 마지막 목록 1, 2, 13, 15, 24, 26, 27, 38이 얻어 진다. 이 알고리듬은 대표적인 분할통치방법이다. 문제를 더 작은 문제들로 나누고 재귀적으로 처리한다. 그리고 통합은 그 작은 문제들의 결과를 결합하여 실현한다. 앞으로 여러번 보게 되겠지만 분할통합에서는 재귀를 매우 효과적으로 리용한다.

병합정렬의 실현을 프로그람 7-4에 주었다. 한 파라메터 mergeSort를 리용하여 4개의 파라메터의 재귀적 mergeSort를 유도할수 있다.

```
/**
 * Mergesort algorithm (driver).
 */
template <class Comparable>
void mergeSort( vector<Comparable> & a )
{
        vector<Comparab1e> tmpArray(   a .size (   ) )
        mergeSortC a, tmpArray, 0, a.size( ) - 1 );
}
```

```
/**
* Internal method that makes recursive calls.
* a is an array of Comparable items.
* tmpArray is an array to place the merged result.
* left is the left-most index of the subarray,
* right is the right-most index of the subarray.
*/
template <class Comparable>
void mergeSort( vector<Comparable> & a,
vector<Comparable> & tmpArray, int left, int right )
{
        if ( left < right )
        {
              int center =( left + right ) / 2;
              mergeSort( a, tmpArray, left, center );
              mergeSort( a, tmpArray, center + 1, right );
              merge( a, tmpArray, left, center + 1, right );
        }
}
```

프로그람 7-4. mergeSort루틴

병합루틴은 까다롭다. 만약 림시적인 배렬을 merge의 매 재귀적호출에 대해 국부적으로 처리되면 임의의 점에서 log$N$개의 림시배렬들을 활성화할수 있다. 마지막검사는 merge가 MergeSort의 마지막행이라는데로부터 거기서는 임의의 점에서 오직 하나의 림시배렬을 활성화할 필요가 있으며 그 림시배렬은 array mergesort구동기로 생성할수 있다는것을 보여 준다. 더우기 림시배렬의 임의의 부분을 리용할수 있는데 입력배렬 a와 같은 부분을 리용한다. 이것을 허용한 개량은 이 절의 마지막에서 서술한다. 프로그람 7-5는 merge루틴을 실현하는 루틴이다.

```
/**
 * Internal method that merges two sorted halves of a subarray.
 * a is an array of Comparable "items.
 * tmpArray is an array to place the merged result.
 * leftPos is the left-most index of the subarray.
 * rightPos is the index of the start of the second half.
 * nghtEnd is the right-most index of the subarray.
 */
template <class Comparable>
void merge( vector<Comparable> & a,
vector<Comparable> & tmpArray,
        int leftPos, int rightPos, int rightEnd )
{
      int leftEnd = rightPos - 1;
```

```
    int tmpPos = leftPos;
    int numElements = rightEnd - leftPos + 1;
    // Main loop
    while( leftPos <= leftEnd && rightPos <= rightEnd )
        if( a[ leftPos ] <= a[ rightPos ] )
            tmpArray[ tmpPos++ ] = a[ leftPos++ ];
        else
            tmpArray[ tmpPos++ ] = a[ rightPos++ ];
    while( leftPos <= leftEnd )       // Copy rest of first half
        tmpArray[ tmpPos++ ] = a[ 1eftPos++ ];
    whi1e( rightPos <= rightEnd )   // Copy rest of right half
        tmpArray[ tmpPos++ ] = a[ rightPos++ ];

    // Copy tmpArray back
    for( int i = 0; i < numElements; i++, rightEnd-- )
        a[ rightEnd ] = tmpArray[ rightEnd ];
}
```

프로그람 7-5. Merge 루틴

1. 병합정렬의 분석

병합정렬은 재귀루틴들을 분석하는데 리용되는 수법의 대표적인 실례이다. 여기에서는 실행시간을 위한 재귀관계를 작성해야 한다. N은 항상 짝수개의 둘로 갈라 진 부분으로 분리되도록 2의 제곱이라고 가정하자. N=1일 때 병합시간은 상수이며 그래서 1로 표시된다. 만일 그렇지 않다면 N개의 수들을 병합정렬하는 시간은 N/2크기의 두개의 재귀적인 병합정렬을 수행하는 시간과 병합하는 시간을 더한것과 같다. 그것은 선형적이다. 다음의 같기식은 이것을 정확히 보여 준다.

$$T(1)=1$$
$$T(N)=2T(N/2)+N$$

이것은 표준적인 재귀관계인데 여러가지 방식으로 풀수 있다. 다음에 두가지 방법을 보여 준다. 첫번째 방법은 재귀관계식을 N으로 나누는것이다. 그 리유는 곧 명백해 진다. 이것은

$$\frac{T(N)}{N}=\frac{T(N/2)}{N/2}+1$$

으로 된다.

이 식은 2의 제곱인 임의의 N에 대하여 타당하다. 따라서

$$\frac{T(N/2)}{N/2}=\frac{T(N/4)}{N/4}+1$$

로 쓸수 있다.

그리고

$$\frac{T(N/4)}{N/4} = \frac{T(N/8)}{N/8} + 1$$

$$\vdots$$

$$\frac{T(2)}{2} = \frac{T(1)}{1} + 1$$

이 식들을 모두 더한다. 이것은 왼쪽에 있는 모든 항들을 더하고 그 결과를 오른쪽에 있는 모든 항들의 값과 같게 설정한다는것을 의미한다. 항 $T(N/2)/(N/2)$은 량변에 다 있으며 따라서 소거된다. 사실상 모든 항들은 량변우에 있으며 소거된다. 이것을 **합의 간단화**(*telescoping a sum*)라고 한다. 모든것을 더한 다음에 마지막결과는

$$\frac{T(N)}{N} = \frac{T(1)}{1} + \log N$$

으로 된다.

왜냐하면 다른 모든 항들이 소거되고 $\log N$식들만 남아 있기때문이며 따라서 이 식의 마지막에 있는 1들은 $\log N$으로 모두 더한다. N을 곱한것이 마지막답이다.

$$T(N)=N\log N+N=O(N\log N)$$

만약 풀기의 시작에서 N으로 나누어 주지 않았다면 합은 간단화되지 못한다. 이것은 N으로 나누는것이 왜 필요한가를 보여 준다.

다른 하나의 방법은 오른쪽에 있는 재귀관계식을 련속적으로 치환하는것이다.

다음과 같은 재귀관계식이 있다.

$$T(N)=2T(N/2)+N$$

$N/2$을 기본등식으로 치환할수 있다는데로부터 즉

$$2T(N/2)=2(2(T(N/4))+N/2)=4T(N/4)+N$$

로부터

$$T(N)=4T(N/4)+2N$$

을 가진다.

다시 $N/4$를 기본등식으로 치환함으로써

$$4T(N/4)=4(2T(N/8)+N/4)=8T(N/8)+8$$

이라는것을 볼수 있다. 따라서

$$T(N)=8T(N/8)+3N$$

을 가진다.

이런 방법을 계속하면

$$T(N)=2^kT(N/2^k)+k \cdot N$$

을 얻는다.

$k=\log N$을 리용하여

$$T(N)=NT(1)+N\log N=N\log N+N$$

을 얻을수 있다.

어느 방법을 선택하겠는가 하는것은 각자의 기호에 관한것이다. 첫번째 방법은 수학적오유가 더 적기때문에 유도함으로써 더 적합한 처리토막을 수행할수 있게 한다. 그러나 그것은 응용에서 일정한 정도의 경험을 요구한다. 두번째 방법은 더 강제적인 방법이다.

$N=2^k$라고 가정한것을 상기하면 분석은 $N=2^k$이 아닌 경우를 처리하도록 개선할수 있다. 결과는 거의 항등적이라고 판명된다. 이것은 일상적으로 그 경우이다.

비록 병합정렬의 실행시간은 $O(\log N)$이라고 하더라도 주기억기에서의 정렬들에 대해서는 힘들게 리용된다. 기본문제는 두개의 정렬된 목록을 병합하는데 선형외부기억기를 리용한다는것이다. 그리고 림시배렬에 다시 복사하는데 소비되는 추가적인 작업은 이 알고리듬전반에서 정렬을 상당히 느리게 한다. 이 복사는 또 다른 재귀준위에서 tmparray의 역할을 정확히 바꿈으로써 피할수 있다. 또한 병합정렬의 변형은 비재귀적으로 실행될수도 있지만 정확한 내부정렬응용에 대해서는 반드시 고속정렬알고리듬을 선택하는것이 좋다. 고속정렬에 대해서는 다음 절에서 서술하게 된다. 병합루틴은 이 장의 마지막에서 보게 되는 많은 외부정렬알고리듬들의 기초로 된다.

제7절. 고속정렬

말그대로 **고속정렬**(*quicksort*)은 실천에서 제일 빨리 정렬하는 알고리듬으로 알려 져 있다. 그의 평균실행시간은 $O(N\log N)$이다. 고속정렬은 매우 빠른데 그것은 주로 매우 엄밀하고 고도로 최적화된 내부순환을 리용하기때문이다. 그것은 최악의 경우에 실행시간이 $O(N^2)$이지만 이것은 약간만 노력하면 지수함수적으로 커지지 않게 할수 있다. 고속정렬알고리듬은 리론적으로는 고도로 최적화될수 있지만 실제로 정확히 코드화할수 없는 알고리듬이라고 여러해동안 평가해 왔지만 리해하기는 간단하고 정확히 증명된다.

고속정렬은 병합정렬과 류사하게 분할통합재귀알고리듬이다.

배렬 S를 정렬하는 알고리듬은 기본적으로 다음과 같은 4개의 단계로 쉽게 이루어 진다.

① S에 속하는 요소수가 0이거나 1이면 처리를 중지한다.

② S에 속하는 어떤 요소 v를 택한다. 이것을 **기준값**(*pivot*)이라고 한다.

③ 분할 $S-\{v\}$(S에서 남아 있는 요소들) 를 두개의 련관이 없는 부분들로 나눈다. 즉

$$S_1 = \{x \in S - \{v\} \mid x \le v,$$
$$S_2 = \{x \in S - \{v\} \mid x \ge v\}$$

④ {고속정렬(S_2)에 이어 v에 이어 진 고속정렬(S_1)}를 돌려 준다.

분할단계에서는 기준값과 같은 요소들을 어떻게 처리하는가에 대하여 애매하게 서술하였는데 이것은 알고리듬설계의 정확성에서 중요한것이다. 훌륭한 실현은 이 경우를 될수 있는데로 효과적으로 조절하는것이다. 직관적으로 보면 기준값보다 작거나 기준값과 같은 절반요소들은 S_1에, 기준값보다 큰 나머지 절반요소들은 S_2에 배치하는것이 좋을것이다.

그림 7-3은 수모임에서 고속정렬의 처리과정을 보여 준다. 기준값은 65로 선택된다. 그 모임에 있는 나머지 요소들은 보다 작은 두개의 모임으로 분할된다.

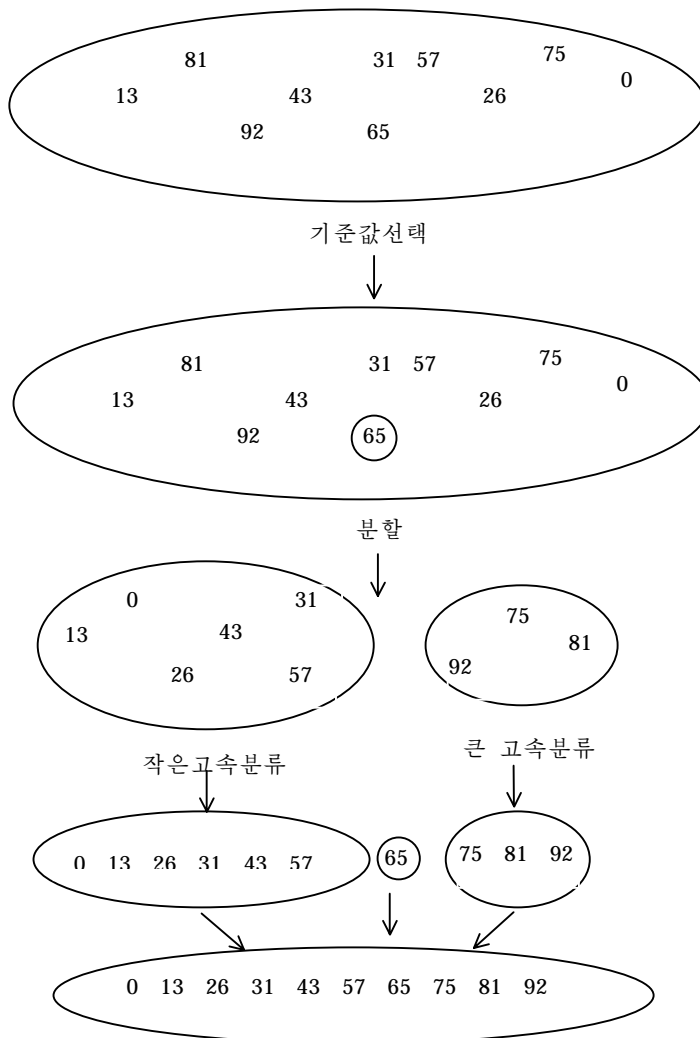



그림 7-3. 고속정렬에 대한 실례

더 작은 수모임을 재귀적으로 정렬하면 0, 13, 26, 31, 43, 57로 된다(재귀규칙 3에 의해서). 큰 수들의 모임도 류사하게 정렬된다. 이러한 방법으로 전체 모임이 쉽게 정렬된다.

이 알고리듬의 처리과정은 명백하지만 병합정렬보다 왜 더 빠른가 하는것은 명백치 않다. 병합정렬처럼 두개의 부분문제들을 재귀적으로 풀고 선형적인 보충작업(단계3)을 요구하지만 병합정렬과는 달리 부분문제들은 같은 치수를 가진다는것이 담보되지 않는다.. 그것은 잠재적으로 좋지 않다. 고속정렬이 더 빠르게 되는것은 분할단계가 실제로 적당하면서도 매우 능률적으로 실행할수 있기때문이다. 이것은 같은 크기의 재귀호출을 담보할수 없다는 부족점을 보상한다. 이에 대하여 지금까지 서술한 이 알고리듬은 구체적인 내용들이 부족하다. 그래서 지금부터 자세히 이야기하겠다. 단계2와 3을 실행하는데는 많은 방법들이 있는데 여기에서 본 방법은 확장분석과 경험적인 연구결과이며 고속정렬을 하면 아주 효과적으로 실행할수 있는 방법이다. 그러나 이 방법으로부터 약간 탈선하면 뜻밖의 나쁜 결과를 야기시킬수 있다.

1. 기준값선택

우에서 서술한것처럼 알고리듬은 기준값요소로 어느 요소가 선택되는가를 걱정하지 않고 작업한다고 하여도 어떤 선택은 명백히 다른것보다 좋은 효과를 가진다..

좋지 못한 방법

일반적으로 경험적인 선택에서는 기준값으로서 첫번째 요소를 리용한다. 이것은 우연적이지만 입력이 먼저 올리 또는 내리순서로 정렬되여 있다면 기준값은 좋지 못한 선택으로 된다. 왜냐하면 모든 요소들이 S_1 혹은 S_2에만 배치되기때문이다. 지어 이것은 재귀호출전반에서 시종일관 발생하게 된다. 실제적인 효과는 첫번째 요소를 기준값으로서 리용하고 입력이 미리 정렬되여 있다고 한다면 고속정렬은 본질상 전혀 아무것도 수행하지 않는데 거기에만도 2차적인 시간이 걸린다는것이다. 이것은 아주 난처한것이다. 더우기 미리 정렬된 입력(혹은 미리 정렬된 큰 토막을 가진 입력)이 빈번히 기준값으로서 첫번째 요소를 리용하는것은 절대적으로 좋지 못한 방법이며 따라서 이러한 선택방법을 즉시 버려야 한다. 또 다른 방법은 기준값으로서 처음 두개의 서로 다른 요소들가운데서 더 큰것을 선택하는것이다. 이것은 순수 첫번째 요소를 택할 때와 같은 나쁜 속성을 가진다. 어느 방법도 택할 필요가 없다.

안전한 방법

안전한 처리과정은 순수 기준값을 우연적으로 선택하는것이다. 란수발생기가 결함을

가지지 않는 이상 이 방법은 일반적으로 완전히 안전하다. 그 결함은 사용자가 생각하는 것처럼 대단한것이 아니다. 왜냐하면 우연적인 기준값이 시종일관 나쁜 분할을 제공한다고 볼수 없기때문이다. 다른 한편 란수발생은 일반적으로 비용이 들며 알고리듬의 나머지의 평균실행시간을 전혀 감소시키지 않는다.

세분할의 중간값

N개의 수들로 이루어 진 부분배렬의 중간값은 $\lceil N/2 \rceil$번째로 제일 큰 수이다. 제일 좋은 기준값은 배렬의 중간에서 선택하여야 한다. 이것은 계산하기가 힘들며 고속정렬을 상당히 느리게 한다. 좋기는 3개의 요소를 우연적으로 선출하여 이 3개 요소의 중간값을 기준값으로 리용하는것이다. 우연성은 그리 좋은것이 아니다. 따라서 일반적으로 왼쪽과 오른쪽, 중간요소의 중간값을 기준값으로 리용한다.

실례로 앞에서처럼 8, 1, 4, 9, 6, 3, 5, 2, 7, 0의 입력에 관해서 왼쪽 요소는 8, 오른쪽 요소는 0 그리고 중간요소는 6이다(중간값은 (왼쪽+오른쪽)/2위치에 있다.). 따라서 기준값 v=6이 된다. 3분할의 중간값을 리용하는것은 명백하게 정렬된 입력인 경우 나쁜 경우를 배제하며(분할은 이 경우에 같게 된다.) 실제적으로 고속정렬의 실행시간을 약 5% 감소시킨다.

2. 분할방법

실천에서 리용되는 여러가지 분할방법들이 있지만 여기서 서술하는 방법은 좋은 결과를 주는것으로 알려 져 있다. 이제 보게 되겠지만 틀리게 혹은 비능률적으로 수행하기는 매우 쉽지만 알려 진 방법을 리용하는것은 안전하다. 처음 단계는 기준값을 마지막 요소와 교환함으로써 기준값을 정렬에 방해가 되지 않게 얻는것이다. i는 첫번째 요소로부터 시작하며 j는 마지막으로부터 바로 앞요소에서 시작한다. 초기입력이 앞에서와 같다면 아래의 그림은 현재의 경우를 보여 준다. 이제 모든 요소들이 다 다르다고 가정하자. 중복되는 요소가 있는데 이에 대한 처리는 후에 취급한다. 제한된 경우처럼 이 알고리듬은 요소들이 모두 같다면 적당하게 수행되여야 한다. 그것은 적당치 않는것을 어떻게 쉽게 처리하는가를 보기로 하자.

분할단계가 수행하려고 하는것은 모든 작은 요소들은 배렬의 왼쪽 부분으로, 모든

큰 요소들은 오른쪽 부분으로 이동하는것이다. 작기와 크기는 물론 기준값과 관계된다. *i*가 *j*의 왼쪽에 있다면 *i*를 오른쪽으로 이동한다. 그때 기준값보다 더 작은 요소들을 뛰여넘는다. *j*는 왼쪽으로 이동하는데 이때 기준값보다 더 큰 요소들을 뛰여 넘으나 *i*와 *j*가 정지되였을 때 *i*는 어떤 큰 요소를 가리키며 *j*는 어떤 작은 요소를 가리킨다. 만약 *i*가 *j*의 왼쪽에 있다면 그 요소들은 교환한다. 좋기는 큰 요소는 오른쪽에, 작은 요소는 왼쪽에 배치하는것이다. 우의 실례에서 *i*는 이동하지 않지만 *j*는 한 위치씩 이동한다. 그 경우는 다음과 같다.

8	1	4	9	0	3	5	2	7	6
↑							↑		
i							j		

그때 *i*와 *j*가 가리키는 요소들을 바꾸며 *i*와 *j*가 사귈 때까지 이런 처리를 반복한다.

첫번째 교환후									
2	1	4	9	0	3	5	8	7	6
↑							↑		
i							j		

두번째 교환하기전									
2	1	4	9	0	3	5	8	7	6
			↑			↑			
			i			j			

두번째 교환후									
2	1	4	5	0	3	9	8	7	6
			↑			↑			
			i			j			

세번째 교환후									
2	1	4	5	0	3	9	8	7	6
					↑	↑			
					j	I			

이 단계에서 i와 j가 사귄다면 그때는 아무런 교환도 진행하지 않는다. 분할의 마지막부분은 기준값을 i가 가리키는 요소와 교환하는것이다.

기준값요소와 교환한후
2 1 4 5 0 3 6 8 7 9   ↑ (6) ↑ (9)   i 기준값

기준값을 마지막단계에서 i와 교환할 때 $p<i$인 매 요소는 작아야 한다는것을 보았다. 이것은 먼저 작은 요소를 포함한 위치 p나 위치 p내에 있는 본래의 큰 요소를 치환하였기때문이다. 류사한 주장은 $p>i$인 요소들은 커야 한다는것을 보여 준다. 여기에서 고찰하여야 할 또 한가지 중요한 내용은 기준값과 같은 요소들을 어떻게 처리해야 하는가 하는것이다. 의문되는것은 i가 기준값과 같은 요소를 만날 때 꼭 정지해야 하며 j도 기준값과 같은 요소를 만날 때 꼭 정지해야 하는가 하는것이다. i와 j는 동일하게 수행되여야 한다. 만약 그렇지 않다면 분할단계는 한쪽으로 치우치기때문이다. 실례로 i는 정지되고 j는 그렇지 않다면 기준값과 같은 모든 요소들은 S_2내에 들어 간다. 좋은 방법을 얻으려면 배렬내에 있는 모든 요소들이 다 같은 경우를 고찰한다. 만약 i와 j가 둘다 정지하면 동등한 요소들사이에서 많은 교환이 있을것이다. 비록 이것은 리용불가능한것으로 보일지라도 명백한 결과는 중간에서 i와 j가 교차할것이라는것이다. 그래서 기준값을 치환할 때 분할은 두개의 거의 같은 부분배렬을 만든다. 병합정렬분석은 그때의 총 실행시간이 $O(N\log N)$이라는것을 말해 준다. 만약 i든 j든 어느것도 정지하지 않는다면 그리고 코드가 배렬의 끝을 지날 때 그것들을 막는다면 어떤 교환도 실행되지 않을것이다. 비록 이것이 좋게 보일지라도 정확한 실행은 기준값을 i가 만나게 되는 마지막위치에서 교환한다. 이것은 마지막 착 앞의 위치일것이다. 혹은 정확히 실행되였다고 보면 마지막일수도 있다. 이것은 홀수부분배렬을 만든다. 만일 모든 요소들이 동등하다면 그 실행시간은 $O(N^2)$이다. 결과 미리 정렬된 입력에 대하여 주목요소로서 첫번째 요소를 리용하는것과 같다. 아무것도 처리하지 않는데 2차시간이 걸린다. 따라서 불필요한 교환들을 처리하고 홀수개의 부분배렬들을 힘들게 다루는것보다 짝수개의 부분배렬들을 만드는것이 좋다는것을 알수 있다. 따라서 i와 j가 기준값과 같은 요소를 만난다면 그 둘은 모두 정지해야 한다. 이것은 이 입력에 대하여 2차시간을 가지지 않도록 하는 4가지 가능성가운데 하나로 된다. 얼핏 보고 동등한 요소로 된 배렬에 대해 걱정하는것은 우둔한것처럼 보인다. 결국 누군가가 5천개의 동등한 요소들을 왜 정렬하려고 하는가? 그러나 고속정렬이 재귀라는것을 상기하자.

10만개의 요소들이 있는데 그중에서 5천개는 동등하다고 하자(더 구체적으로는 정렬열쇠들이 같은 복잡한 요소들). 종국적으로 고속정렬은 이 5천개의 요소들에 대하여서만

재귀호출을 할것이다. 그러면 실제적으로 5천개의 항등요소들이 효과적으로 정렬될수 있다는것을 확인하는것이 중요하게 될것이다.

3. 작은 배렬

매우 작은 배렬($N \leq 20$)에 대한 고속정렬은 삽입정렬처럼 잘 수행되지 않는다. 더우기 고속정렬은 재귀적이기때문에 이런 경우들이 자주 생긴다. 일반적인 해결방도는 작은 배렬에 대해 고속정렬을 재귀적으로 리용하지 않는것이지만 대신 작은 배렬에 대해서는 삽입정렬과 같은 능률적인 정렬알고리듬을 리용한다. 이 방법을 리용하면 실행시간을 약 15% 절약할수 있다(그렇게 하는것은 어떤 차단도 주지 않는다.).비록 5와 20사이의 임의의 절단은 류사한 결과를 생성한다고 하더라도 좋은 절단범위는 N=10이다. 이것은 또한 하나 혹은 두개의 요소만 있을 때 세 요소의 중간값을 취하는것과 같은 좋지 못한 경우를 없앤다.

4. 실제적인 고속정렬루틴

고속정렬에 대한 구동프로그람을 프로그람 7-6에서 보여 준다.

```
/**
  * Quicksort algorithm (driver).
  */
template <class Comparable>
void quicksort( vector<Comparable> & a)
{
      quicksort( a, 0, a.size( ) - 1 );
}
```

프로그람 7-6. 고속정렬에 대한 구동프로그람

루틴들의 일반형식은 배렬과 정렬하려는 배렬의 범위(왼쪽과 오른쪽)를 통과하게 하는것이다. 취급되는 첫번째 루틴은 기준값선택이다. 이에 대한 제일 쉬운 방법은 적당히 a[left], a[right], a[center]를 위치적으로 정렬하는것이다. 이것은 3개가운데서 제일 작은것을 a[left]에 넣는다는 특별한 우점을 가진다. 이 방법은 분할단계를 어쨌든 아무데나 설정하게 할것이다. 제일 큰것은 a[right]안에 넣는데 그것은 또한 정확한 위치이다. 왜냐하면 그것은 기준값보다 더 크기때문이다. 따라서 기준값을 a[right - 1]에 위치하고 i와 j를 분할단계에서 left+1과 right - 2로 초기화할수 있다. 또 다른 리득은 a[left]가 기준값보다 더 작기때문에 그것은 j에 대해 감시요소처럼 작용할것이라는것이다. 따라서 끝을 지

나는 j에 대하여 걱정할 필요가 없다. i는 기준값과 같은 요소들에서는 정지하게 되므로 a[right - 1]에 기준값을 보관하는것은 i에 대한 감시요소를 제공하는것으로 된다.

프로그람 7-7에 있는 코드는 서술한 모든 효과로 세 분할의 중간값을 처리한다. 그것은 다만 a[left], a[senter], a[right]를 실지로 정렬하지 않는 방법으로 기준값을 계산하는것이 약간 비효률적이라고 볼수 있지만 뜻밖에도 이것은 나쁜 결과들을 초래한다(련습문제 7-46을 보시오.).

```
/**
 *Return median of left, center, and right.
 * Order these and hide the pivot.
 */
template <class Comparable>
const Comparable & median3( vector<Comparab1e> & a, -int left, int right )
{
        int center = ( left + right ) / 2;
        if( a[ center ] < a[ left ] )
             swap( a[ left ], a[ center ] );
        if( a[ right ] < a[ left ] )
             swap( a[ left ], a[ right ] );
        if( a[ right ] < a[ center ] )
             swap( a[ center ], a[ right ] );

             // Place pivot at position right - 1
        swap( a[ center ], a[ right - 1 ] );
        return a[ right - 1 ];
}
```

프로그람 7-7. 세 분할의 중간값을 처리하는 코드

고속정렬루틴의 실제 핵심부는 프로그람 7-8에 있다. 그것은 분할과 재귀호출을 포함하고 있다. 이 실행에는 가치 있는 여러가지 특징들이 있다. 행 3은 i와 j를 1이상의 자체의 정확한 값들로 초기화한다. 따라서 거기에는 고찰하여야 할 어떠한 특수한 경우는 없다. 이 초기화는 세분할의 중간값이 일부 측면들에서 효과를 가진다는 사실에 관계된다. 이 프로그람은 만약 사용자가 단순한 기준값선택방법을 변화시키지 않고 그것을 리용하려고 하면 처리되지 않는다. 그것은 i와 j는 적당치 않는 위치에서 시작하며 j에 대한 감시요소가 더이상 없기때문이다.

8행에 있는 교환은 때때로 속도를 높일 목적으로 명확히 작성된다. 알고리듬이 빨리 수행되도록 하기 위해 콤파일러가 이 코드를 직결로 콤파일하게 할 필요가 있다. 많은 콤파일러들은 Swap가 inLine리용으로 선언된다면 이것을 자동적으로 수행하지만 만일 그렇지 않다면 자동적으로 수행하지 않을수 있다.

```
        /**
          * Internal quicksort method that makes recursive calls.
          * Uses median-of-three partitioning and a cutoff of 10.
          * a is an array of Comparable items.
          * left is the left-most index of the subarray.
          * right is the right-most index of the subarray.
          */
        template <class Comparable>
        void quicksort ( vector<Comparable> & a, int left, int right )
        {
/* 1*/      if( left + 10 <= right )
             {
/* 2*/          Comparable pivot = median3( a, left, right );
                    // Begin partitioning
/* 3*/          int i = left, j = right - 1;
/* 4*/          for( ; ; )
                 {
/* 5*/                while( a[ ++i ] < pivot ) { }
/* 6*/              whi1e( pivot < a[ - -j ]) { }
/* 7*/              if( 1 < j )
/* 8*/                  swap( a[ 1 ], a[ j ] );
                else
/* 9*/                  break;
     }
/*10*/          swap( a[ i ], a[ right - 1 ] );      // Restore pivot
/*11*/          quicksort( a, left, i - 1 );       // Sort small elements
/*12*/          quicksort( a, i + 1, right );       // Sort large elements
             }
           else // Do an insertion sort on the subarray
/*13*/          insertionSort( a, left, right );
         }
```

프로그램 7-8. 기본고속정렬루틴

마지막으로 5행과 6행은 고속정렬이 왜 그렇게 빠른가를 보여 준다. 알고리듬의 내부순환고리는 증가/감소(1만큼씩 빨리), 검사, 이행으로 구성된다. 병합정렬할 때에는 어떤 특별한 조작이 없다. 이 코드는 여전히 까다롭다. 3~9행까지를 프로그램 7-9에 있는 명령문들로 치환해 보자. 만일 a[i] = a [j] = pivot이면 무한순환이 되기때문에 이것은 처리되지 않는다.

```
/* 3*/          int i = left + 1, j = right - 2;
/* 4*/          for( ; ; )
                {
/* 5*/              whi1e( a[ i ] < pivot ) i++;
/* 6*/              whi1e( pivot < a[ j ] ) j--;
```

```
/* 7*/                 if( i < j )
/* 8*/                      swap( a[ i ], a[ j ] );
                         else
/* 9*/                      break;
                    }
```

프로그램 7-9. 고속정렬에 대한 약간한 변화
(그래서 그 알고리듬을 파괴한다.)

병합정렬처럼 고속정렬도 재귀적이다. 그리고 그의 분석은 재귀식을 풀것을 요구한다. 우연적인 기준값과 작은 배렬에 대해서는 어떤 차단도 없다는것을 전제로 하고 고속정렬에 대한 분석을 진행하여 보자. 병합정렬에서처럼 $T(0)=T(1)=1$을 가지게 된다.

5. 고속정렬의 분석

고속정렬에 대한 실행시간은 2개의 재귀호출의 실행시간 + 분할에서 소비하는 실행시간과 같다(기준값선택은 단지 상수시간만을 취한다.). 이것은 기본고속정렬관계식

$$T(N)=T(i)+T(N-i-1)+cN \tag{7-1}$$

을 준다.

여기서 $i=|S_1|$는 S_1에 있는 요소의 개수다. 이제 3가지 경우로 고찰하자.

최악의 경우 분석

기준값은 언제나 제일 작은 요소이다. 이때 i=0이고 $T(0)$=1이라는것을 무시한다면 재귀는

$$T(N)=T(N-1)+cN, \quad N>1 \tag{7-2}$$

이다. 식 7-2를 반복하여 리용하면 간단화한다. 따라서

$$T(N-1)=T(N-2)+c(N-1) \tag{7-3}$$

$$T(N-2)=T(N-3)+c(N-2) \tag{7-4}$$

$$\cdots$$

$$T(2)=T(1)+c(2) \tag{7-5}$$

이다.

전체 식들을 모두 더하면 앞에서 고찰한것처럼

$$T(N)=T(1)+c\sum_{i=2}^{N} i = O(N^2) \tag{7-6}$$

으로 된다.

제일 좋은 경우 분석

제일 좋은 경우에 기준값은 중간에 있다. 수학적으로 간단화하기 위해 두개의 부분배렬을 매개가 정확히 본래크기의 절반이라고 가정한다. 그리고 이것은 약간 지나치게 평가한다고 하여도 다만 큰O응답에 관심을 가지므로 접수될수 있다.

$$T(N)=2T(N/2)+c(N) \tag{7-7}$$

식 7-7의 량변을 N으로 나누자.

$$\frac{T(N)}{N}=\frac{T(N/2)}{N/2}+c \tag{7-8}$$

이 식을 리용하여 간단화를 한다.

$$\frac{T(N/2)}{N/2}=\frac{T(N/4)}{N/4}+c \tag{7-9}$$

$$\frac{T(N/4)}{N/4}=\frac{T(N/8)}{N/8}+c \tag{7-10}$$

$$\cdots$$

$$\frac{T(2)}{2}=\frac{T(1)}{1}+c \tag{7-11}$$

식 7-7로부터 식 7-11까지 모든 식들을 더한다. 이때 그가운데 $\log N$이 있다는것을 주의하자.

$$\frac{T(N)}{N}=\frac{T(1)}{1}+c\log N \tag{7-12}$$

이것은

$$T(N)=cN\log N+N=O(N\log N) \tag{7-13}$$

으로 된다.

이것은 병합정렬과 정확히 같은 분석이라는것에 주의를 돌리면 같은 결과를 얻는다는것을 강조한다.

평균경우의 분석

이것은 가장 어려운 부분이다. 평균경우 S_1에 대하여 매개 크기들이 거의 같고 그래서 확률 $1/N$을 가진다고 하자. 이 가정은 실제로 기준값선택과 분할방법에 대해 타당하지만 기타 다른것에 대해서는 타당치 않다. 부분배렬들의 우연성을 보존하지 않는 분할방법들은 이 분석을 리용할수 없다.

재미 있는것은 이 방법들이 실천적으로 오래동안 실행되여 결과가 나오는것처럼 보

이는것이다.

이 가정에 관하여 $T(i)$의 평균값은 $T(N-i-1)$이라는데로부터 $(1/N)\sum_{j=0}^{N-1}T(j)$이다.

그러면 식 7-1은

$$T(N)=\frac{2}{N}\left[\sum_{j=0}^{N-1}T(j)\right]+cN \tag{7-14}$$

이다.

만일 식 7-14에 N를 곱하면 그것은

$$NT(N)=2\left[\sum_{j=0}^{N-1}T(j)\right]+cN^2 \tag{7-15}$$

로 된다.

문제를 간단하게 하기 위해 합기호를 없앤다. 여기에서는 하나이상의 식으로 간단화할수 있다는것을 지적한다.

$$(N-1)T(N-1)=2\left[\sum_{j=0}^{N-2}T(j)\right]+c(N-1)^2 \tag{7-16}$$

식 7-16을 7-15에서 덜면

$$NT(N)\text{-}(N\text{-}1)T(N\text{-}1)=2T(N\text{-}1)+2cN\text{-}c \tag{7-17}$$

을 얻는다. 항들을 재정돈하고 오른쪽에서 의의가 없는 c를 무시하면

$$NT(N)=(N+1)T(N\text{-}1)+2cN \tag{7-18}$$

을 얻는다.

지금 $T(N\text{-}1)$에서만 $T(N)$에 대한 식을 가진다. 이 식은 다시 간략화할수 있지만 식 7-18은 적당치 않은 형식이다. 식 7-18을 $N(N+1)$으로 나누어서

$$\frac{T(N)}{N+1}=\frac{T(N-1)}{N}+\frac{2c}{N+1} \tag{7-19}$$

을 얻는다.

$$\frac{T(N-1)}{N}=\frac{T(N-2)}{N-1}+\frac{2c}{N} \tag{7-20}$$

$$\frac{T(N-2)}{N-1}=\frac{T(N-3)}{N-2}+\frac{2c}{N-1} \tag{7-21}$$

$$\frac{T(2)}{3}=\frac{T(1)}{2}+\frac{2c}{3} \tag{7-22}$$

식 7-19부터 7-22까지 더하면

$$\frac{T(N)}{N+1}=\frac{T(1)}{2}+2c\sum_{i=3}^{N+1}\frac{1}{i} \quad (7\text{-}23)$$

을 얻는다.

합은 약 $\log_e(N-1)+\gamma-3/2$이다. 여기서 $\gamma\approx0.577$은 오일레르의 수라고 한다. 따라서

$$\frac{T(N)}{N+1}=O(\log N) \quad (7\text{-}24)$$

이며 이것은 다시

$$T(N)=O(N\log N) \quad (7\text{-}25)$$

로 된다.

비록 이 분석은 복잡한것처럼 보이지만 사실은 그렇지 않다. 즉 그 단계들은 이미전에 사용자가 일부 재귀관계식들에서 보아 왔기때문에 자연스러운것이다. 이러한 분석은 앞으로도 진행될수 있다. 우에서 서술된것처럼 고도로 최적화된 알고리듬이 분석되였다. 그리고 이 결과는 복잡한 재귀와 더 높은 수준의 수학을 동반하므로 매우 어렵다. 또한 같은 요소들의 효과는 자세하게 분석되였으며 표시된 코드가 정확히 수행된다는것을 알수 있다.

6. 선형기대시간의 선택알고리듬

제1장과 제6장에서 고찰한것과 같은 선택문제(selection problem)를 풀기 위하여 고속정렬을 수정할수 있다. 우선권대기렬을 리용함으로써 k번째 제일 큰(제일 작은) 요소를 $O(N+k\log N)$시간에 찾을수 있다는것을 상기하자. 중간값을 찾기 위한 특수한 경우에 있어서 이것은 $O(N\log N)$알고리듬을 주게 된다.

그 배렬을 $O(N\log N)$시간에 정렬할수 있기때문에 선택을 위한 더 좋은 시간한계를 얻을것을 기대할수 있다. 모임 S에서 k번째로 제일 작은 0요소를 찾는 이 알고리듬은 거의 고속정렬과 같다. 사실상 처음 세개의 단계들은 같다. 이 알고리듬을 **고속선택**(*quickSelect*)이라고 한다.

① $|S|=1$이면 $k=1$이며 그 결과로서 S에 속하는 그 요소를 돌려 준다. 만일 작은 배렬들에 대한 차단이 리용되고 있고 $|S|\leqq$CUTOFF라면 S를 정렬하고 k번째 제일 작은 요소를 되돌려 준다.

② 기준값요소를 선택한다. $v\in S$

③ 고속정렬에서와 마찬가지로 분할 $S-\{v\}$을 S_1과 S_2로 분할하자.

④ $k\leqq|S_1|$라면 k번째 제일 작은 요소는 S_1에 속해야 한다.

이 경우에 고속선택 (S_1, k)에로 되돌린다. 만일 $k=1+|S_1|-1$이라면 기준값은 k번째로 제일 작은 요소이며 그것을 선택결과로 돌려 줄수 있다.

만약 그렇지 않다면 k번째 제일 작은 요소는 S_2에 놓이며 그것은 S_2에서 (k-|S_1|-1)번째 로 제일 작은 요소이다. 재귀호출을 처리하고 고속선택 (S_2, k-|S_1|-1)을 돌려 준다.

고속정렬과는 반대로 고속선택은 재귀호출을 두번할 대신에 오직 한번만 리용한다. 고속선택의 최악의 경우는 고속정렬일 때와 항상 같으며 $O(N^2)$이다. 이것은 고속정렬의 최악의 경우가 S_1과 S_2중의 하나가 빌 때이며 따라서 고속선택은 실지 재귀호출을 보관하지 않는다. 그러나 평균실행시간은 $O(N)$이다. 그 분석은 고속정렬과 류사하며 련습문제로 남긴다.

고속선택의 실행은 지어 추상적인 서술이 암시하는것보다 더 쉽다. 이것을 실행하는 코드를 프로그람 7-10에서 보여 준다. 알고리듬이 끝날 때 k번째 제일 작은 요소는 k−1 위치에 있다(그것은 배렬이 첨수 0에서부터 시작하기때문이다. 이것은 원래순서를 파괴하는데 만일 이것이 바랄수 없다면 복사가 이루어 져야 한다.

```
        /**
         * Internal selection method that makes recursive calls.
         * Uses median-of-three partitioning and a cutoff of 10.
         * Places the kth smallest item in a[k-1].
         * a is an array of Comparable items.
         * left is the left-most index of the subarray.
         * right is the right-most index of the subarray.
         * k is the desired rank (1 is minimum) in the entire array.
         */
        template <class Comparable>
        void quickSe1ect( vector<Comparab1e> & a, int left, int right, int k )
            {
/* 1*/      if( left + 10 <= right )
              {
/* 2*/          Comparable pivot = median3( a, left, right );

                  // Begin partitioning
/* 3*/          int i = left, j = right - 1;
/* 4*/          for( ; ; )
                  {
/* 5*/              whi1e( a[ ++i ] < pivot ) { }
/* 6*/              whi1e( pivot < a[ --j ] ) { }
/* 7*/              if( i < j )
/* 8*/                  swap( a[ i ], a[ j ] );
                      else
/* 9*/                  break;
                }
/*10*/          swap( a[ i ], a[ right - 1 ] );      // Restore pivot

                      // Recurse; only this part changes
```

```
/*11*/          if( k <= i )
/*12*/                  quickSe1ect( a, left, i - 1, k );
/*13*/          else if( k > i + 1 )
/*14*/                  quickSe1ect( a, i + 1, right, k );
            }
            else // Do an insertion sort on the subarray
/*15*/          insertionSort( a, left, right );
      }
```

프로그람 7-10. 기본고속선택루틴

세분할의 중간값방법을 리용하는것은 거의 무시해도 좋은 최악의 경우의 기회를 만들수 있다. 그렇지만 기준값을 심중하게 선택함으로써 2차의 최악의 경우를 제거하고 $O(N)$알고리듬을 담보한다. 이것을 수행하는데 포함되는 전체적인 비용은 상당하며 따라서 전체적인 알고리듬은 대체로 리론적인 흥미를 가진다. 제10장에서 선택을 위한 선형시간의 최악의 경우에 대한 알고리듬을 시험할것이다. 그리고 실천적으로 더 빠른 선택알고리듬을 가지게 하는 기준값을 선택하는 흥미 있는 기법도 고찰하게 될것이다.

제8절. 간접정렬

고속정렬은 고속정렬이고 Shell정렬은 Shell정렬이다. 그러나 이 알고리듬에 기초한 함수형판들을 직접 실현하는것은 정렬된 comparable객체들이 클 때는 때때로 비능률적일수 있다. 문제는 Comparable들을 정돈할 때 comparable객체들을 반복적으로 복사하는 전체적인 비용을 보상하여야 하는것이다(그에 대한 operator=함수를 호출함으로써). 이것은 만일comparable객체들 복사하기에는 크고 어렵다면 비용이 커질수 있다.

원리적으로 문제의 해답은 단순하다. comparable에 대한 지적자배렬을 생성하고 그 지적자들을 재정돈한다. 일단 그 요소들이 배치되는 곳을 안다면 중간복사의 전체적인 비용이 없이 그것들을 거기에 배치할수 있다. 이를 잘 수행하려면 in-situ순렬이라는 알고리듬이 요구된다. C++언어로 이것을 실현하려면 새로운 문법들이 요구된다.

알고리듬의 첫째 단계에서 지적자배렬을 만든다. a는 정렬하려는 배렬이라고 하고 p는 지적자배렬이라고 하자. 초기에 p[i]는 a[i]에 기억된 대상에 대한 지적자일것이다. 다음 순차성을 결정하는데서 지시된 객체의 값을 리용하여 p[i]를 정돈한다. 배렬 a에 속하는 대상들은 전혀 이동하지는 않지만 배렬 p에 속하는 지적자들은 재배치된다. 그림 7-4는 지적자배렬들을 정렬단계후에 어떻게 보겠는가를 보여 준다.

여기에서는 여전히 배렬 a를 재정돈하여야 한다. 이것을 수행하는 제일 단순한 방법

은 Comparable의 두번째 배렬을 선언하는것인데 그것을 copy라고 한다. 그때 정확한 정렬순서를 Copy에 작성하고 다음 Copy에서 다시 a에로 작성한다. 이것을 수행하는 비용은 여분배렬과 2*N*개의 Comparable 복사의 전체이다.

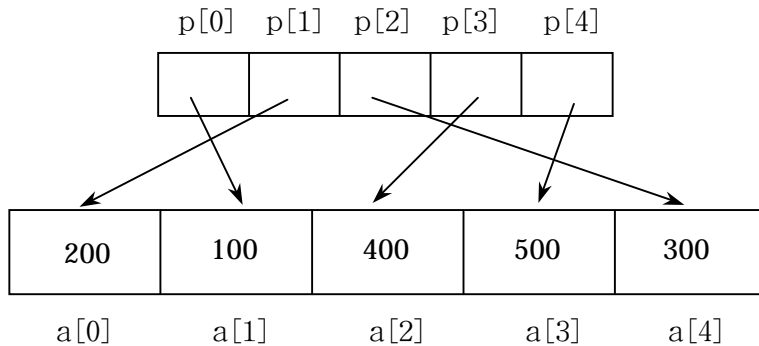



그림 7-4. 정렬에 대한 지적자배렬의 리용

알고리듬은 잠재적으로 중요한 문제를 가진다. copy를 리용함으로써 공간요구는 두배로 된다. *N*이 크고(크지 않다면 삽입정렬을 리용한다.) Comparable객체의 크기도 크다(크지 않다면 지적자실현을 리용하는것을 걱정하지 않을것이다.)고 가정하자.따라서 콤퓨터의 기억기한계내에서 연산하리라는것을 기대할수 있다. 그렇지만 지적자들의 여분벡토르를 리용하는것은 기대할수 있더라도 리용할수 있는 comparable객체의 여분벡토르는 반드시 기대할수 없다. 따라서 여분배렬을 재정렬함이 없이 배렬 a를 적당히 재배렬할 필요가 있하여야 한다.

copy를 리용하겠는가 하는데서 두번째로 제기되는것은 총 2*N*의 Comparable복사들을 리용하는것이다. 비록 이것은 원래알고리듬이상에서의 개선이지만 그 알고리듬을 더 개선할수 있는 방법을 고찰하게 된다. 특히 Comparable복사를 3*N*/2이상 리용하지는 않으며 거의 모든 입력들에 대해 단지 *N*보다 조금 많이 리용한다. 공간을 절약할뿐아니라 시간도 절약할것이다.

코드를 작성하기전에 무엇이 필요하겠는가에 대하여 일반적인 사상에 대하여 보기로 하자.이미 앞에서 그것을 수행하였다.우리가 무엇을 해야 하겠는가에 대하여 보기 위하여 *i*=2로부터 시작하자. p[2]가 a[4]를 가리킨다는데로부터 a[4]를 a[2]로 이동하여야 한다. 먼저 a[2]을 보관해야 한다. 왜냐하면 후에 그것을 정확한 위치에 놓을수 없기때문이다. 다음 tmp=a[2]이고 그다음 a[2]=a[4]이다. a[4]가 a[2]로 이동할 때 a[4]로 무엇이든지 이동할수 있는데 a[4]는 본질적으로 비여 있다. p[4]를 검사함으로써 정확한 명령문이 a[4]=a[3]이라고 본다. 다음 a[3]에로 무엇이든지 이동할 필요가 있다. p[3]은 a[2]을 가리킨다는데로부터 a[2]를 거기로 이동하기 쉽다는것을 안다. 그런데 a[2]는 이 재정돈

의 시작에서 덧씌여 졌다. 즉 그의 본래값은 tmp에 있기때문에 a[3]=tmp로 끝낸다. 이 처리는 i=2로부터 시작하고 p배렬에 따라 순환렬 2, 4, 3, 2를 형성한다는것을 보여 준다.

이 순환렬은

```
tmp     = a[ 2 ];
a[ 2 ] = a[ 4 ];
a[ 4 ] = a[ 3 ];
a[ 3 ] = tmp;
```

에 대응한다.

오직 4개의 Comparable복사와 하나의 기억기의 림시Comparable을 리용하여 3개 요소들을 재정돈하였다. 실제로 이 방법을 이미 앞에서 보았다. 삽입정렬의 제일 안쪽 고리는 tmp대상안에 현재의 요소 a[i]를 보관한다. 다음 많은 요소들을 오른쪽으로 하나씩 이동하기 위해 a[j] = a[j-1]를 할당한다.

마지막으로 원래의 요소를 배치하기 위하여 a[j]=tmp를 할당한다. 여기서는 하나씩 밀어 준는것을 제외하고는 정확히 같은것을 수행하며 재배치가 어떻게 되는가를 알려 주기 위하여 p를 리용하고 있다. 같은 밀기알고리듬은 2진더미의 insert에서도 리용되고 있다.

일반적으로는 재정돈되는 순환들이 수집된다. 그림 7-4에는 두개의 순환이 있다. 하나는 두개의 요소를 포함하고 다른 하나는 3개를 포함하고 있다. 길이가 L인 순환의 재정돈은 L+1개의 Comparable복사들을 리용한다. 길이가 1인 순환들은 이미 정확히 배치된 요소들을 표현하며 그래서 어떤 복사도 리용하지 않는다. 이것은 앞의 알고리듬에 비하여 개선으로 된다. 그것은 이미 정돈된 배렬은 어떤 Comparable복사들도 초래하지 않기 때문이다.

N개의 요소들로 이루어 진 배렬에서 C_L은 길이가 L인 순환들의 개수라고 하자. Comparable복사의 총수 M은

$$M=N-C_1+(C_2+C_3+\ldots C_N) \tag{7-26}$$

으로 된다.

발생할수 있는 제일 좋은 경우는 어떤 Comparable복사도 없는것인데 그것은 길이가 1인 N개의 순환들이 있기때문이다(즉 매 요소는 정확히 배치된다.). 발생할수 있는 최악의 경우는 길이가 2인 N/2개의 순환들을 가진다. 이 경우에 식 7-26은 M=3N/2개의 comparable복사가 수행된다. 이것은 입력이 2, 1, 4, 3, 6, 5 등과 같은 경우에 발생할수 있다. M에 대해 기대되는 값은 얼마인가? 련습문제는 M이 $N-2+H_N$이라고 보여 주고 있다.

이 정렬알고리듬을 수행하기 위해 먼저 p에 기억되는 객체의 형에 대해 우선 클라스형판 pointer를 준다. 이것을 프로그람 7-11에서 보여 준다.

프로그람 7-12에서 largeObjectSort라는 이름을 가지는 함수형판을 써주었다. 코드는 지적자조작과 련관된 C++의 여러가지 개선된 개념들을 리용한다.

```
/* 1*/    template <class Comparable>
/* 2*/    class Pointer
/* 3*/    {
/* 4*/        public:
/* 5*/            Pointer( Comparable *rhs = NULL ) : pointee( rhs ) { }
/* 6*/
/* 7*/            bool operator<( const Pointer & rhs ) const
/* 8*/            { return *pointee < *rhs.pointee; }
/* 9*/
/*10*/            operator const Comparable * ( ) const
/*11*/            { return pointee; }
/*12*/        private:
/*13*/            Comparable *pointee;
/*14*/    };
```

프로그램 7-11. Comparable에 대한 지적자를 보관하는 클라스

```
/*15*/    template <class Comparable>
/*16*/    void 1argeObjectSort( vector<Comparab1e> & a
/*17*/    {
/*18*/        vector<Pointer<Comparable> > p( a.sizeC ) );
/*19*/        int i, j, nextj;
/*20*/
/*22*/        for( i = 0; 1 < a.s-ize( ); i++ )
/*22*/            p[ i ] = &a[ i ];
/*23*/
/*24*/        quicksort( p );
/*25*/
/*26*/            // Shuffle items in place
/*27*/        for( i = 0; i < a.size( ); i++ )
/*28*/            if( p[ i ] !=&a[ i ] )
/*29*/            {
/*30*/                Comparable tmp = a[ i ];
/*31*/                for( j = i; p[ j ] != &a[ i ]; j = nextj )
/*32*/                {
/*33*/                    nextj = p[ j ] - &a[ 0 ];
/*34*/                    a[ j ] = *p[ j ];
/*35*/                    p[ j ] = &a[ j ];
/*36*/                }
/*37*/                a[ j J = tmp;
/*38*/                p[ j ] = &a[ j ];
/*39*/            }
/*40*/    }
```

프로그램 7-12. 큰 객체들을 정렬하는 알고리듬

1. 작업하지 않는 벡토르 <Comparable*>

기본방법은 vector<Comparable*>로써 지적자들의 배렬 p를 선언하고 그 지적자들을 재정돈하기 위해 quicksort(p)를 호출하는것이다. 그런데 이것은 처리되지 않는다. 문제는 quicksort에 대한 형판인수가 Comparable*이며 그래서 2개의 Comparable*형들을 비교할수 있는 <연산자를 요구하는것이다. 이와 같은 어떤 연산자는 지적자형들을 위해 존재하지만(제1장 제5절 1을 상기하시오.) 이 연산자의 결과는 지시된 Comparable내에 보관된 값들을 가지고 아무것도 처리하지 않는다. 앞으로 이 성질을 무시할수 없다.

2. 적응지적자클라스

문제의 해결은 새로운 클라스형판 pointer를 정의하는것이다. pointer는 자료성원으로서 Comparable에 대한 지적자를 보관한다. 그다음 pointer형에 대한 비교연산자를 준다. 이것은 프로그람 1-22에 있는 Employee클라스에서 수행하던것과 류사하다.

자료성원 pointer는 13행에 있는 은페부에서 소개된다. pointer클라스에 대한 구축자는 pointer에 대한 초기값 (혹은 NULL)을 요구하는데 이것은 5행에서 보여 준다.

지적자의 성질을 밀봉하는 클라스들을 때때로 적응지적자클라스(smart pointer class)라고 한다. 이 클라스는 초기값이 제공되지 않았을 때 그자체를 자동적으로 NULL로 초기화하기때문에 보통 지적자보다 적응성이 더 높다.

3. < 연산자의 다중정의

<연산자를 실현하는것은 개념적으로 단순하다. 지적되고 있는 Comparable객체들에 < 연산자를 방금 적용하였다. 이것은 재귀적인 론리가 없다는데 주목하자. 클라스 Pointer 내에서 7행에 있는(형판) operator < 는 두개의 Pointer형들을 비교하며 8행에서의 호출은 두개의 Comparable형들을 비교한다.

4. 별표 * 에 의한 지적자의 역참조

8행에서 별표(*)를 가진것은 무엇인가? 별표(*)는 C++에서 지적자비참조연산자이다.

만약 ptr가 어떤 객체에 대한 지적자이라면 *ptr는 지시되고 있는 객체에 대한 동의어이다. 다시말하여 만약 ptr가 대상 obj를 가리킨다면 *ptr는 obj와 같다.

*ptr의 값은 obj의 값이며 *ptr에 대한 모든 변화들은 obj에서 변화들을 일으킨다. 이 연산자와 주소연산자 &는 임의의 다른것보다 C++에서 더 많은 문제를 야기시킨다. 그러

나 그것들은 언제나 리용할 가치가 있다. 여기서 그 리용은 불가피하다.

5. 형변환연산자의 다중정의

10행은 기묘한 C++문장을 아주 좋게 보여 준다. 이것은 형변환연산자인데 특히 이 방법은 Pointer<Comparable>로부터 Comparable*로의 형변환을 정의한다. 그 실현은 단순한데 11행에서 Pointer를 되돌려 준다. 이것은 그 지적자를 얻는데 리용할수 있다. 비록 getPointer와 같은 성원함수를 리용할수 있다고 해도 이 형변환은 largeObject Sort알고리듬을 단순하게 한다.

6. 어디서나 찾아 볼수 있는 암시적인 형변환

C++는 강한 형정의언어이다. 코드에는 다음과 같은 형들이 있다.

```
a[ i ]          comparable
&a[ i ]         comparable*
p[ i ]          Pointee<comparable>
```

그러므로 &a[i]와 p[i]는 형호환성이 없다고 본다. 코드내에는 다음의 4가지 다른 경우들을 포함하여 호환성이 없는 많은 실례들이 있다.

```
/*22*/      p[ i ] = &a[ i ];
/*28*/        if(p[ i ] ! = &a[ i ])
/*33*/                next j = p[ j ] - &a[ 0 ];
/*34*/                a[ j ] = *p[ j ];
```

강한 형정의를 하는것은 무엇을 발생하겠는가? 10행에서 형변환연산자를 주고 pointer설치자에 대한 explicit를 리용하지 않음으로써 형호환을 할수 없었다. 특이한것들을 찾아 보자.

22행은 pointer설치자가 explicit가 아니기때문에 작업한다. p[i]는 pointer <comparable> 이라는데로부터 오른변 역시 1이여야 한다. 가령 1이 아니더라도 pointer<comparable>지적자를 리용하여 설치자를 리용하는것은 림시로 만들어 질수 있으며(이것은 암시적인것이다.) explicit가 제외되기때문에 배경적인 설치가 허용된다.

28행은 pointer<comparable>과 comparable*와 비교하기 위해 Operator!=를 리용한다. 이 연산자는 존재하지 않는다. 더우기 암시적인 변환(10행에 있는 형변환연산자를 리용하여)은 림시적인 comparable*을 만드는데 리용될수 있다. 결과 operator!=는 수행된다.

33행은 C++의 특징의 또 다른 토막이다. 그에 대해서 후에 고찰한다. 여기서 10행에 있는 형변환연산자는 p[j]로부터 림시적인 comparable*을 만든다. 34행는 비참조연산자를

pointer<comparable>에 적용하려고 시도한것이다. 그러나 이 연산자는 그 형에 대해 정의되지 않는다(그것은 파부하는 될수 있지만 수행하지는 않는다.). 그런데 10행에서의 형변환연산자는 p[j]로부터 림시적인 comparable*을 만드는것으로써 기일을 절약할수 있다.

7. 모호성을 낳는 쌍방향암시적변환

이 형변환들은 작업할 때 많이 쓰이지만 그것들은 기대할수 없는 문제들이 생기게 할수 있다. 실례로 7행과 8행에서 정의된 연산자 < 외에 연산자 operator!=도 제공하였다고 가정하자(이것은 지나친 과장은 아니다. 확대된 탐색나무들의 일부는 operator< 에 보충하며 opreator!=에 의거한다.).

28행은 더이상 콤파일러되지 않는데 이것은 모호성이 생겼기때문이다. 지금 p[i]를 comparable*로 변환하고 초기지적자변수들에 대해 정의된 !=연산자를 리용하든지 혹은 설치자를 리용하여 &a[i]를 pointer<comparable>로 승격시키고 다음 pointer <comparable>에 의해 정의되는 !=를 리용할수 있다.

이와 같은 난처한 립장에서 벗어 나기 위한 많은 방식들은 있지만 이것때문에 어떤 일반적이 아닌 클라스내에서 쌍방향암시적변환을 정의해서는 결코 안된다는것을 말해 둔다. 만약 늘 explicit를 리용하거나 혹은 형변환연산자들을 리용하지 않는다면 이 문제는 없을것이다.

8. 합법적인 지적자덜기

최종적인 비결은 33행에 있다. p1과 p2는 같은 배렬내에 있는 두개의 요소들을 지시한다면 p1-p2는 int처럼 그들사이의 거리이다. 따라서 p[j]-&a[0]은 p[j]가 가리키는 객체의 첨수이다.

제9절. 정렬의 일반적인 아래한계

비록 정렬알고리듬이 $O(N\log N)$을 가진다 해도 이것을 훌륭히 수행할수 있다는것은 명백치 않다. 이 절에서는 비교만 리용하는 모든 알고리듬이 최악의 경우에 $\Omega(N\log N)$번의 비교를 요구하며 따라서 병합정렬과 더미정렬은 상수인자내에서 최량이다. 증명은 오직 비교만을 리용하는 정렬알고리듬에 대해서 평균 $\Omega(N\log N)$번의 비교가 요구된다는것을 보여 주는데로 확장할수 있다. 이것은 고속정렬이 상수인자내에서 평균적으로 최량이라는것을 증명하게 된다.

특별히 다음과 같은 결과를 증명한다. 즉 오직 비교만을 리용하는 정렬알고리듬은 최악의 경우에는 $\lceil \log(N!) \rceil$번의 비교를 요구하며 평균적으로는 $\log(N!)$의 비교를 요구한다. 모든 정렬알고리듬이 이런 경우에 작업해야 한다는데로부터 모든 N개의 요소들은 서로 다르다고 가정한다.

1. 결정나무

결정나무는 아래한계를 증명하는데 리용된 추상화이다. 문맥상으로 결정나무는 2진나무이다. 매 마디는 요소들사이에서 이루어 지는 비교들의 모순이 없는 가능한 경우들의 모임을 나타낸다. 비교들의 결과는 나무의 끝이다.

그림 7-5에 있는 결정나무는 요소 a, b, c를 정렬하는 알고리듬을 나타낸다.알고리듬의 초기상태는 뿌리에 있다(용어 <상태>와 <마디>를 구별없이 리용한다.). 뿌리에서는 어떤 비교도 진행되지 않는다. 그래서 순서를 붙인 모든것이 정당하다 .이 특정의 알고리듬이 수행하는 첫번째 비교는 a와 b의 비교이다. 두개의 결과는 두개의 가능한 상태를 이끌어 낸다. 만약 $a<b$이면 오직 3개의 가능성만이 남는다. 만약 알고리듬이 마디 2에 도달하면 그것은 a와 c를 비교할것이다. 다른 알고리듬들은 서로 다른것을 수행할수 있다. 즉 다른 알고리듬은 다른 결정나무를 가진다. 만약 $a>c$이면 알고리듬은 상태 5로 들어 간다. 모순이 없는 하나의 경우만 있다는데로부터 알고리듬은 끝나서 정렬이 완성되였다고 알린다.

$a<c$이면 알고리듬은 이것을 할수 없다. 왜냐하면 두개의 가능한 경우들이 있는데 그것은 어느것이 정확한가를 확인할수 없기때문이다. 이 경우에 알고리듬은 하나이상의 비교를 요구한다.

비교들만 리용하여 정렬하는 매 알고리듬은 결정나무에 의해 표현될수 있다. 물론 그것은 극히 작은 입력크기들에 대한 나무를 묘사하는데는 적당할수 있다. 정렬알고리듬에서 리용되는 비교들의 개수는 제일 깊은 잎의 깊이와 같다. 우리의 경우에 이 알고리듬은 최악의 경우에 3개의 비교를 리용한다. 리용된 비교의 평균수는 잎들의 평균깊이와 같다.

결정나무가 크다는데로부터 거기서 일부 긴 경로들이 있어야 한다는것이 나온다. 웃한계를 증명하기 위해 보여 줄 필요가 있는 모든것은 기본나무의 몇가지 속성이다.

보조정리 7-1.
T는 깊이가 d인 2진나무라고 하자. 그때 T는 기껏해서 2^d개의 잎을 가진다.

증명:
증명은 귀납법으로 한다. 만약 d=0이라면 기껏해서 한개 잎이 있다. 그래서 정리는

성립한다. $d\neq 0$이라면 잎이 될수 없고 잎과 왼쪽, 오른쪽 부분나무가 될수 없는 뿌리를 가진다. 여기서 왼쪽, 오른쪽 부분나무의 깊이는 기껏해서 d-1이다. 귀납법가정에 의하여 그것들은 기껏해서 총 2^d개의 잎을 가지며 2^{d-1}개의 잎을 가질수 있다. 이것으로 정리는 증명된다.

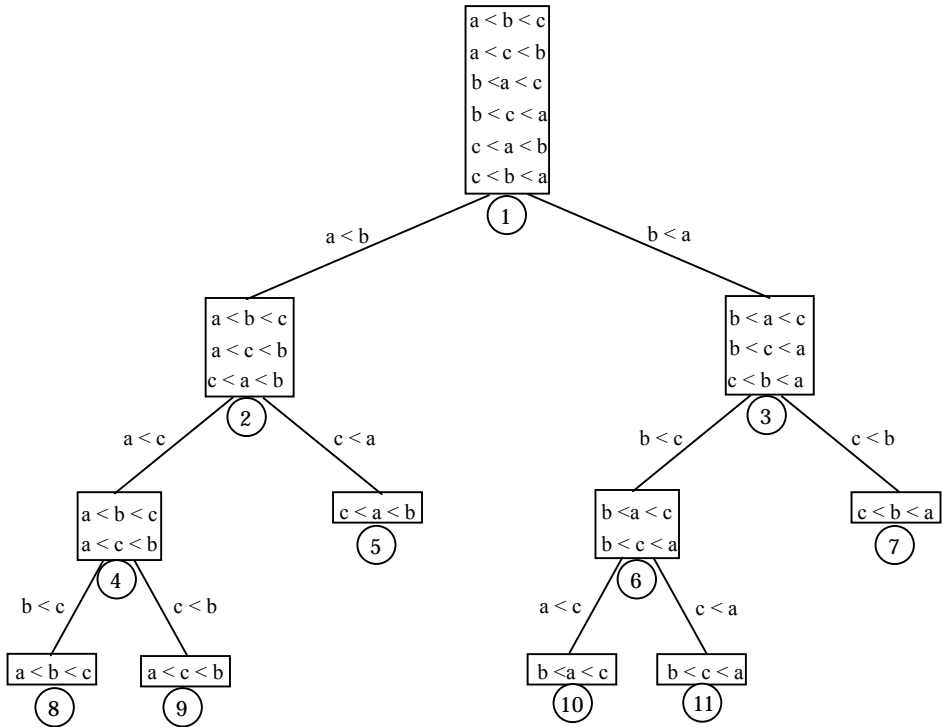



그림 7-5. 3요소삽입정렬을 위한 결정나무

보조정리 7-2.

L개의 잎을 가진 2진나무는 적어도 $\lceil \log L \rceil$인 깊이를 가져야 한다.

증명:

앞의 보조정리로부터 즉시 나온다.

정리 7-6.

요소들사이의 비교들만 리용하는 모든 정렬알고리듬은 최악의 경우에 적어도 $\lceil \log(N!) \rceil$번의 비교를 요구한다.

증명:

N개 요소들을 정렬하는 결정나무는 $N!$개의 잎을 가져야 한다. 이 정리는 앞의 보

조정리로부터 나온다.

정리 7-7.
요소들사이의 비교만 리용하는 모든 비교알고리듬은 $\Omega(N\log N)$번의 비교를 요구한다.

증명:
앞의 정리로부터 log(N!)번의 비교가 요구된다.

$$\log(N!) = \log(N(N-1)(N-2)\ldots(2)(1)$$
$$= \log(N + \log(N-1) + \log(N-2) + \ldots + \log 2 + \log 1$$
$$\geqq \log N + \log(N-1) + \log(N-2) + \ldots + \log(N/2)$$
$$\geq \frac{N}{2}\log\frac{N}{2}$$
$$\geq \frac{N}{2}\log N - \frac{N}{2}$$
$$= \Omega(N\log N)$$

이러한 형태의 아래한계에 대한 설명은 최악의 경우의 결과를 증명하는데 리용될 때 때때로 정보리론적아래한계라고 한다.

일반정리는 다음과 같은것을 보여 준다. 즉 구별되는 P개의 서로 다른 경우가 있고 질문이 YES/NO형태이라면 일부 경우에 문제를 풀기 위한 알고리듬들에 의해 늘 $\lceil \log P \rceil$개의 질문들이 요구된다. 임의의 비교에 토대한 정렬알고리듬에 대하여 평균경우 실행시간에 대한 류사한 결과를 증명하는것은 가능한 일이다.

이 결과는 다음과 같은 보조정리(련습문제로서 남겨 둔다.)들에 의해서 암시된다. 즉 L개의 잎을 가진 임의의 2진나무는 적어도 $\log L$인 평균깊이를 가진다.

제10절. 바께쯔정렬

비록 앞절에서 비교만을 리용하는 모든 일반정렬알고리듬이 최악의 경우에 $\Omega(N\log N)$시간을 요구한다는것을 증명하였지만 그것은 여전히 일부 특수경우에는 선형시간으로 정렬하는것이 가능하다는것을 상기하자.

단순한 실례가 바께쯔(Bucket)정렬이다. Bucket정렬에 대해 작업하자면 여분의 정보를 리용할수 있어야 한다. 입력 $A_1, A_2, \cdots, A_N$은 M보다 더 작은 정의옹근수들로만 이루어진다(이에 대한 명확한 확장은 가능하다.). 만일 이것이 사실이라면 알고리듬은 단순하다. count라고 불리우는 배렬(크기는 M이고 모두 0으로 초기화된다.)을 가진다고 하자. count

는 M개의 세포 혹은 Bucket를 가진다. 그 Bucket는 초기에 비여 있다. A_i를 읽을 때 count[A_i]는 하나씩 증가된다. 모든 입력이 읽히워 진후에 정렬된 목록에 대한 표현을 출력하면서 count배렬을 훑는다. 이 알고리듬은 $O(M+N)$을 가지며 증명은 련습으로 남긴다. M이 $O(N)$이라면 총 합은 $O(N)$이다.

비록 이 알고리듬이 아래한계를 위반하는것처럼 보인다 하더라도 그것은 단순한 비교들보다 더 위력한 연산을 리용하기때문에 수행되지 않는다는것이 판명된다.적당한 바께쯔를 증명함으로써 알고리듬은 본질적으로 단위시간내에 M-방식비교를 수행한다. 이것은 확장할수 있는 하쉬에서 리용된 방법과 류사하다 (제5장 제6절). 이것은 명백히 아래한계가 증명된 모형에는 없다.

그렇지만 이 알고리듬은 아래한계를 증명하는데서 리용된 모형의 타당성을 확인한다. 실제로 그 모형은 강한 모형이다. 왜냐하면 일반용정렬알고리듬은 입력의 형에 대한 가정을 만들수는 없지만 순서 붙은정보에 기초한 판단들만은 이루어 져야 하기때문이다.자연적으로 리용할수 있는 여분정보가 있다면 더 능률적인 알고리듬을 찾는것이 기대되여야 한다(만약 그렇지 않다면 여분정보는 점점 없어져 간다는데로부터).

비록 바께쯔정렬은 리용하는 보통의 알고리듬과 너무도 많이 류사한것처럼 보이지만 입력에는 오직 작은 옹근수들인 경우가 있다는것이 판명되였다. 그래서 고속정렬과 같은 방법을 리용하는것을 없앤다.

제11절. 외부정렬

지금까지 검사한 모든 알고리듬은 입력이 주기억기에 어울릴것을 요구한다. 그런데 입력이 주기억에 어울리기에는 너무도 큰 응용프로그람도 있다. 이 절에서는 외부정렬알고리듬을 설명하며 외부정렬알고리듬들은 매우 큰 입력들을 처리하도록 설계된다.

1. 새 알고리듬의 필요성

대부분의 내부정렬알고리듬들은 기억기를 직접 주소화할수 있는 우점을 가진다. 쉘정렬은 요소 a[i]와 a[i-h_k]를 한시간단위내에서 비교한다. 더미정렬은 요소 a[i]와 2[I×2+1]를 같은 시간내에 비교한다.

세 분할중간값을 가진 고속정렬은 a[left], a[senter], a[right]를 일정한 시간내에 비교할것을 요구한다. 입력이 테프우에 있다면 이 모든 연산들은 자기의 능률을 잃는다. 왜냐하면 테프우에 있는 요소들은 오직 순차적으로만 호출될수 있기때문이다.

가령 자료가 플로피우에 있다고 하더라도 여전히 플로피의 회전과 플로피머리의 이

동에 요구되는 지연때문에 효률이 실제적으로 손실된다.

외부호출이 실제로 얼마나 느린가를 보자면 크기는 하지만 주기억에 어울리기에는 너무 크지 않는 우연적인 파일을 만든다.

그 파일을 기억기에 읽어 들여서 그것을 능률적인 알고리듬을 리용하여 정돈한다. 입력을 읽는데 걸리는 시간은 그 입력를 정돈하는 시간과 비교된다는것이 확실하다(가령 정렬은 $O(N\log N)$연산이며 그 입력을 읽는것은 오직 $O(N)$이라고 하더라도).

2. 외부정렬에 대한 모형

대규모기억장치들의 넓은 변화는 내부정렬보다도 장치에 관계되는 외부정렬을 한다. 고찰하게 되는 알고리듬들은 테프우에서 작업한다. 그 테프는 가장 제한된 기억매체이다.

테프우의 요소에 대한 호출을 정확한 위치에 테프를 감는 방법으로 수행하기때문에 테프들은 순차적인 차례로만 호출될수 있다(둘중의 어느 한 방향으로).

정렬을 진행하기 위해서 적어도 테프구동기를 가진다고 가정하자.

능률적으로 정렬하기 위해서는 2개의 구동기가 요구된다. 3번째의 구동기는 문제를 단순하게 한다. 만약 한개의 테프구동기만 존재한다면 어려울것이다. 모든 알고리듬은 $\Omega(N^2)$의 테프호출을 요구한다.

3. 간단한 알고리듬

기본외부정렬알고리듬은 병합정렬에 의한 병합알고리듬을 리용한다. 4개의 테프 T_{a1}, T_{a2}, T_{b1}, T_{b2}을 가진다고 가정하자. 그것은 2개의 입력과 2개의 출력테프들이다. 알고리듬에서 론점에 따라 a와 b테프들은 입력테프든지 출력테프로 된다. 자료는 초기에 T_{a1}에 있다고 가정하자. 더 나가서 내부기억기는 한번에 M개의 레코드들을 보존(그리고 정렬)할수 있다고 가정한다. 지연적인 첫번째 걸음을 입력테프로부터 한번에 M개의 레코드를 읽어 들이고 그 레코드들을 내부적으로 정렬하고 다음 정렬된 레코드들을 T_{b1}과 T_{b2}들중의 어느 하나에 쓰는것이다. 정렬된 레코드들의 매 모임은 *run*을 호출한다. 이것이 수행될 때 모든 테프들을 다시 감는다. 쉘정렬에 대한 실례로서 같은 입력을 가진다고 가정하자.

T_{a1}	81	94	11	96	12	35	17	99	28	58	41	75	15
T_{a2}													
T_{b1}													
T_{b2}													

M=3이라면 *run*들이 만들어 진후 테프는 다음의 그림에서 지적된 자료를 포함한다.

T_{a1}			
T_{a2}			
T_{b1}	11 81 94	17 28 99	15
T_{b2}	12 35 96	41 58 75	

그러면 T_{b1}과 T_{b2}는 *run*그룹을 포함한다. 매 테프로부터 첫번째 *run*을 취하고 그것들을 조합한 다음 그 결과를 T_{a1}우에 쓴다. 두개의 정렬된 목록을 병합하는것은 단순하다는 것을 상기하자. 즉 병합은 T_{b1}과 T_{b2}가 전진할 때 수행된다는데로부터 어떤 기억기도 거의 요구하지 않는다. 그다음 매 테프로부터 다음의 *run*을 취하고 이것들을 병합하여 그 결과를 T_{a2}에 쓴다. T_{b1}이나 T_{b2}가 빌 때까지 T_{a1}과 T_{a2}을 서로 치환하면서 이 처리를 계속한다. 이 시점에서 둘다 다 비든지 혹은 하나의 *run*을 남겨 둔다. 후자의 경우에 이 *run*을 적당한 테프에 복사한다. 4개의 모든 테프를 다시 감고 입력으로서 a테프를 리용하고 출력으로 b테프를 리용하면서 같은 단계를 반복한다. 이것은 4M개의 *run*을 줄것이다. 길이가 N인 한개의 *run*을 얻을 때까지 처리를 계속 한다. 이 알고리듬은 $\lceil \log(N/M) \rceil$ + 최초의 *run*만들기단계를 요구한다. 만일 매개가 128byte인 1000만개의 레코드와 4Mbyte의 외부기억기를 가진다면 첫번째 단계는 320개의 *run*을 만들것이다. 그러면 정렬을 완성하는데 9이상의 단계를 요구한다. 실례는 $\lceil \log 13/3 \rceil = 3$ 이상의 단계를 요구한다.

그것은 다음의 그림에서 보여 준다.

T_{a1}	11 12 35 81 94 96	15
T_{a2}	17 28 41 58 75 89	
T_{b1}		
T_{b2}		

T_{a1}	
T_{a2}	
T_{b1}	11 12 17 28 35 51 58 75 91 94 96 99
T_{b2}	15

T_{a1}	11 12 15 17 28 35 41 58 75 81 94 96 99
T_{a2}	
T_{b1}	
T_{b2}	

4. 여러길병합

여분의 테프를 가지고 있다면 입력을 정렬하는데 요구되는 단계수는 감소시킬수 있다고 기대한다. 이것은 기본(2방식)병합을 k방식병합으로 확장함으로써 수행한다.

T_{a1}			
T_{a2}			
T_{a3}			
T_{b1}		11 81 94	41 58 75
T_{b2}		12 35 96	15
T_{b3}		17 28 99	

T_{a1}	11 12 17 28 35 81 94 96 99	
T_{a2}	15 41 58 75	
T_{a3}		
T_{b1}		
T_{b2}		
T_{b3}		

T_{a1}	
T_{a2}	
T_{a3}	
T_{b1}	11 12 15 17 28 35 51 58 75 81 94 96 99
T_{b2}	
T_{b3}	

두개의 *run*의 병합은 매 *run*의 시작부에로 매 입력테프를 감아 놓는것으로써 수행한다. 그러면 더 작은 요소를 찾아서 출력테프우에 놓으며 해당한 입력테프는 전진한다. 만약 k개의 입력테프들이 있다면 이 방법은 같은 방식으로 작업한다. 이때 오직 차이는 그것이 k개의 요소들가운데서 제일 작은것을 찾는것보다 약간 더 복잡하다는것이다. 우선권대기렬을 리용하여 이 요소들가운데서 제일 작은것을 찾을수 있다. 출력테프우에 쓰는 다음 요소를 얻으려면 deleteMin연산을 수행한다. 해당한 입력테프는 전진하며 그리고 입력테프우에 있는 *run*이 아직 완성되지 않았다면 새로운 요소를 우선권대기렬내에 insert한다. 전과 같은 실례를 리용하여 입력을 3개의 테프에 분배한다.

그러면 정렬을 완성하는데 3방식병합의 2이상의 통과가 필요하다. 초기의 *run*을 만든 후에 k방식병합을 리용할때 요구되는 통과수는 $\lceil \log_k(N/M) \rceil$이다. 왜냐하면 *run*들은 매 통과에서 k배만큼 크게 얻어 지기때문이다.

우의 실례에 대해 공식은 $\lceil \log_3(13/3) \rceil = 2$ 라는데로부터 확인된다.

만약 10개의 테프를 가진다면 k=5이며 앞절에서의 큰 실례는 $\lceil \log_5 320 \rceil = 4$ 통과를 요구한다.

5. 여러단계병합

앞절에서 개발된 k-방식병합방법은 $2k$개의 테프를 리용할것을 요구한다. 이것은 일부 응용프로그람들에서는 금지될수 있다. k+1개의 테프만을 가지고도 성공할수 있다. 실례로서 오직 3개의 테프만을 리용하여 2-방식병합을 실행하는것을 보여 준다.

3개의 테프 T_1, T_2, T_3과 T_1상에 있는 입력파일을 가진다고 가정하자. 이때 T_1은 34개의 *run*들을 생성할것이다.

한가지 선택은 T_2와 T_3이 매개우에 17개의 *run*을 놓는것이다. 다음 이 결과를 T_1우에서 병합할수 있다. 이때 17개의 *run*을 가진 하나의 테프가 얻어 진다. 문제는 모든 *run*이 한 테프우에 있다는데로부터 또 다른 병합을 수행하기 위해 이것들중의 일부 *run*들을 T_2우에 놓아야 한다는것이다. 이것을 수행하는 론리적방식은 T_1에 있는 첫 8개의 *run*들을 T_2우에 복사하고 병합을 진행하는것이다. 이것은 수행하는 매 통과들에 대해 여분의 절반단계를 첨부하는 효과를 가진다. 또 다른 방법은 초기의 34개의 *run*들을 똑같지 않게 나누는것이다. 21개의 *run*을 T_2에, 13개의 *run*은 T_3에 놓는다고 가정하자. 다음 T_3이 빌 때까지 T_1우에 있는 13개의 *run*들을 병합한다. 이 시점에서 T_1과 T_3을 다시 감고 13개의 *run*을 가진 T_1과 8개의 *run*을 가진 T_2을 T_3우에서 병합한다. 다음 T_2가 빌 때까지 8개의 *run*을 병합한다. 이때 T_1우에는 5개의 *run*이, T_3우에는 8개의 *run*이 남아 있다. 다음 T_1과 T_3을 병합하고 …이런 식으로 계속한다. 다음의 표는 매 단계이후 매 테프우에 있는 *run*들의 개수를 보여 준다

	상수적인 실행	**T_3+ T_2 한다음**	**T_1+ T_2 한다음**	**T_1+ T_3 한다음**	**T_2+ T_3 한다음**	**T_1+ T_2 한다음**	**T_1+ T_3 한다음**	**T_2+ T_3 한다음**
T_1	0	13	5	0	3	1	0	1
T_2	21	8	0	5	2	0	1	0
T_3	13	0	8	3	0	2	1	0

*run*들의 초기분배에는 큰 차이가 있다. 실례로 만일 22개의 *run*이 T_2우에, 12개의 *run*이 T_3 우에 배치된다면 첫번째 병합후에 T_1우에는 12개의 *run*이, T_2우에는 10개의 *run*이 얻어 진다. 또 다른 병합후에 T_1에는 10개의 *run*이, T_3우에는 2개의 *run*이 있다. 이 시

점에서 상태가 천천히 얻어 진다. 왜냐하면 오직 T_3이 비기전에 *run*의 두개 모임을 조합할수 있기때문이다. 다음 T_3이 8개의 *run*을 가지고 T_2는 2개의 *run*을 가진다. 다시 *run*의 두개의 모임을 병합할수 있다.

이때 6개의 *run*을 가진 T_1과 2개의 *run*을 가진 T_3이 얻어 진다. 3개이상의 통과후에 T_2는 2개의 *run*을 가지며 다른 테프들은 빈다. 하나의 *run*을 또 다른 테프에 복사해야 하며 다음에 병합을 끌낼수 있다.

첫번째 분배가 최적이라는것이 판명된다. *run*들의 개수가 피보나치수 F_N이라면 그것들을 나누는 제일 좋은 방식은 그것들을 두개의 피보나치수들 $F_{N\text{-}1}$과 $F_{N\text{-}2}$로 나누는것이다.

그렇지 않다면 피보나치수까지의 *run*들의 개수를 얻기 위해 거짓 *run*들을 가진 테프를 덧붙이는것이 필요하다.

테프우에 *run*들의 초기모임을 배치하는 상세한 방법을 련습으로서 남긴다. 이것을 *k*−방식병합으로 확장할수 있다. *k*−방식병합에서 분배를 위해 *k*번째 순번의 피보나치수들을 요구한다. 거기서 *k*번째의 피보나치수는 다음과 같이 정의된다.

$$F^{(k)}(N)=F^{(k)}(N\text{-}1)+F^{(k)}(N\text{-}2)+\ldots+F^{(k)}(N\text{-}k)$$

여기서 적당한 초기조건은 다음과 같이 결정한다.

$$F^{(k)}(N)=0,\quad 0\leq N\leq k\text{-}2,\quad F^{(k)}(k\text{-}1)=1$$

6. 치환선택

고찰하게 될 마지막항목은 *run*들에 대한 만들기이다. 지금까지 리용하여 온 방법은 제일 단순한 가능성들이다. 즉 가능한만큼 많은 레코드들을 읽어서 그것들을 정렬하고 그 결과를 어떤 테프에 썼다. 이것은 첫번째 레코드가 출력테프에 씌여 지자마자 그것을 리용할 기억기가 또 다른 레코드에 대해 리용할수 있게 된다는것을 깨달을 때까지 이것은 가장 좋은 방법처럼 보인다. 만약 출력테프우에 있는 다음의 레코드가 방금 출력한 레코드보다 더 크다면 그것은 *run*에 포함될수 있다.

이런 고찰을 리용하여 *run*들을 생성하는 알고리듬을 줄수 있다. 이 기술을 일반적으로 **치환선택**(replacement selection)이라고 한다. 초기에 *M*개의 레코드들을 기억기내에 읽어 들여 우선권대기렬내에 배치한다. 제일 작은 레코드가 출력테프에 씌여 지면서 deleteMin을 수행한다. 입력테프로부터 다음 레코드를 읽는다. 만약 그것이 방금 쓴 레코드보다 더 크다면 그것을 우선권대기렬에 첨부할수 있다. 만약 그렇지 않다면 그것은 현재 사용중으로 갈수 없다. 우선권대기렬이 하나의 요소정도로 작으므로 이 새로운 요소를 *run*이 완성될 때까지 우선권대기렬의 무효공간에 기억시킬수 있으며 다음 *run*을 위하여 그 요소를 리용할수 있다. 무효공간에서 요소를 정렬하는것은 더미정렬에서 수행하는

것과 류사하다. 이것을 우선권대기렬의 크기가 0(이 시점에서 *run*은 초과된다.)으로 될 때까지 계속한다. 모든 무효공간내에 있는 요소들을 리용하여 새로운 우선권대기렬을 만듦으로써 새로운 *run*을 시작한다. 표 7-4는 리용해 온 간단한 실례에 대한 *run*건설을 보여 준다. 이때 M=3이다. 무효요소들은 *로서 지적된다.

표 7-4. 실행구축의 실례

	더미배렬에 있는 3개의 요소들			출력	다음 요소읽기
	h[1]	h[2]	h[3]		
실행 1	11	94	81	11	96
	81	94	96	81	12*
	94	96	12*	94	35*
	96	35*	12*	95	17*
	17	35*	12*	실행끝	더미재구축
실행 2	12	35	17	12	99
	17	35	99	17	28
	28	99	35	28	58
	35	99	58	35	41
	41	99	58	41	15*
	58	99	15*	58	테프끝
	99		15*	99	
			15*	실행끝	더미재구축
실행 3	15			15	

이 실례에서 치환선택은 오직 3개의 *run*들을 생성하며 정렬에 의해 얻어 진 5개의 *run*들과 비교된다. 이로 하여 3-방식병합은 둘대신에 한통과로 끝난다.

만약 입력이 우연적으로 분할된다면 치환선택은 평균길이가 2M인 *run*들을 생성한다는것을 보여 줄수 있다.

큰 실례들에 대하여 320개의 *run*대신에 16개의 *run*을 생각한다.

그래서 5-방식병합은 4통과를 요구한다. 이 경우에 비록 행운이 주어 져서 125개의 *run* 혹은 그보다 적게 가지고 있다 하더라도 통과는 절약하지 못한다.

외부정렬은 그렇게 오래 걸린다는데로부터 절약된 매 통과는 실행시간에서 효과적인 차이가 생기게 한다.

우리가 본것처럼 그것은 치환선택이 표준알고리듬보다 더 좋지 않게 수행될수 있다. 그렇지만 입력은 자주 정렬되거나 거의 시작에서 정렬된다. 이 경우에 치환선택은 극소수의 매우 긴 *run*들을 생성한다. 이 종류의 입력은 외부정렬들에 대해 공통이며 치환선택을 매우 가치 있게 한다.

요약

대부분의 일반적인 내부정렬응용프로그람들에서는 삽입정렬이나 쉘정렬 혹은 고속정렬들중의 어느 하나를 선택하는것이 좋다. 이중에서 어느것을 리용하겠는가 하는 결심은 대체로 입력의 크기에 관계된다. 표 7-5는 여러가지 입력크기에 대하여 매 알고리듬을 실행시켜 얻은 실행시간을 보여 준다.(매우 느린 콤퓨터상에서)

자료는 N개의 옹근수들이 우연적으로 바꾸어 선택되도록 하였다. 주어 진 시간은 다만 정렬을 위한 실제시간이다. 그림 7-2에서 주어 진 코드는 삽입정렬을 위해 리용되였다. 쉘정렬은 (제7장 제4절에 있는 코드를 리용하였다.)Sedgewick의 증분들을 가지고 실행하도록 수정하였다. 말그대로 100만개에 대한 정렬에 토대할 때(100부터 2500만까지의 크기범위에 있다.) 이 증분들을 가진 쉘정렬의 기대되는 실행시간은 $O(N^{7/6})$으로 추측된다. 더미정렬루틴은 제7장 제5절과 같다.

표 7-5. 각이한 정렬알고리듬에 대한 비교(단위는 초)

N	삽입정렬	쉘정렬	더미정렬	고속정렬	고속정렬(최적)
10	0.00044	0.00041	0.00057	0.00052	.00046
100	0.00675	0.00171	0.00420	0.00284	.00244
1000	0.59564	0.02927	0.05565	0.03153	.02587
10000	58.864	0.42998	0.71650	0.36765	.31532
100000	NA	5.7298	8.8591	4.2298	3.5882
1000000	MA	71.164	104.68	47.065	41.282

고속정렬에는 두가지 변종이 있다. 첫번째는 단순한 주목요소방법을 리용하며 차단을 수행하지 않는다. 이때 입력는 우연적이다. 두번째는 세개의 중간분할과 10개의 차단을 리용한다. 그이상의 최량화들도 가능하다. 함수를 리용할대신 행에서 3개의 중간루틴을 코드화하였으며 비재귀적으로 고속정렬을 작성할수 있다. 실행하는데 매우 보편적인 코드에 대한 일부 다른 최량화들도 있다. 물론 아쎔불러를 리용할수도 있다. 모든 루틴들을 능률적으로 코드화하기 위한 정당한 시도를 해왔다. 그러나 그 성능은 콤퓨터에 따라 변할수 있다.

고속정렬에 대한 높은 최량화형식은 입력의 크기가 매우 작을 때 쉘정렬만큼 빠르다. 고속정렬의 개선된 형식은 최악의 경우에 여전히 $O(N^2)$을 가지지만 (자그마한 실례를 들어 련습하여 보시오.)나타나는 이 최악의 경우는 하나의 요인만을 가지는것이 아니므로 무시할수 있다. 만약 사용자가 많은 량의 자료를 정렬할 필요가 있다면 고속정렬을 선택하는것이 좋다.

그러나 결코 쉬운방식을 택하지 말아야 하며 첫번째 요소를 기준값요소로서 리용하

지 말아야 한다. 입력이 우연적이라고 가정하는것은 안전하지 않다. 만일 이에 대하여 걱정하지 않으려면 쉘정렬을 리용하는것이 좋다. 쉘정렬은 작은 성능오유를 주지만 특별히 간단한 경우에는 그대로 접수할수 있다. 그의 최악의 경우는 오직 $O(N^{4/3})$인데 발생하는 그 최악의 경우는 마찬가지로 무시할수 있다.

비록 명백히 맞물린 내부순환을 가진 $O(N\log N)$알고리듬이라고 해도 더미정렬은 쉘정렬보다 느리다. 알고리듬에 대한 마지막시험은 자료를 이동하기 위해 더미정렬은 두개의 비교를 진행한다는것을 보여 준다. Floyd에 의해 암시된 개선은 본질적으로는 오직 한번의 비교만을 가지고 자료를 이동하지만 이 개선의 실행은 어쨌든 코드를 더 길게 한다. 특별한 코드화의 노력이 속도증가에 가치가 있는가 없는가하는것은 독자들의 판단에 맡긴다(련습문제 7-51). 삽입정렬은 작거나 혹은 대체적으로 정렬된 입력들에 대해서만 리용한다.

병합정렬은 포함하지 않는다. 왜냐하면 그의 성능은 주기억기의 정렬에 대해 고속정렬만큼 좋지 않으며 그것은 코드화하기가 단순하지 않기때문이다. 그렇지만 병합이 외부정렬의 중심적인 방법이라는것을 보았다.

련습문제

7-1. 삽입정렬을 리용하여 렬 3, 1, 4, 1, 5, 9, 2, 6, 5를 정렬하시오.

7-2. 모든 요소들이 다 같다면 삽입정렬의 실행시간은 얼마로 되는가?

7-3. 초기에 정돈되여 있지 않는 요소 a[i]와 a[i+k]를 치환한다고 가정하자. 적어서 1이고 많아서 $2k-1$개의 반전들이 제거된다는것을 증명하시오.

7-4. 증분 {1, 3, 7}을 리용하여 입력 9, 8, 7, 6, 5, 4, 3, 2, 1에 대하여 쉘정렬의 실행결과를 보여 주시오.

7-5. ㄱ. 두개의 증분렬 {1,2}을 리용할 때 쉘정렬의 실행시간은 얼마인가?

ㄴ. 임의의 N에 대하여 쉘정렬이 $O(N^{5/3})$시간으로 실행하는것과 같은 3-증분렬이 존재한다는것을 보여 주시오.

ㄷ. 임의의 N에 대하여 쉘정렬이 $O(N^{3/2})$시간으로 실행되는것과 같은 6-증분렬이 존재한다는것을 보여 주시오.

7-6. *ㄱ. 쉘정렬의 실행시간이 임의의 옹근수 c에 대해 증분형식 1, c, $c^2, \ldots, c^i$을 리용할 때 $\Omega(N^2)$을 증명하시오.

**ㄴ. 이 증분들에 대해 평균실행시간은 $O(N^{3/2})$이라는것을 증명하시오.

***7-7.** k-k정렬된 파일이 h-정렬이라면 그것은 k정렬을 남긴다는것을 증명하시오.

****7-8.** 히바드에 의해서 암시된 증분렬을 리용할 때 쉘정렬의 실행시간은 최악의 경우에 $\Omega(N^{3/2})$이라는것을 증명하시오. 암시: 요소들이 모두 0이거나 1일 때 쉘정렬에 대한 특수경우를 고찰하여 한계를 증명할수 있다. i는

$h_t, h_{t-1}, \cdots, h_{\lfloor t/2 \rfloor+1}$인 선형조합으로서 표현된다면 $a[i]$=1로 설정하시오. 만약 그렇지 않다면 0으로 설정하시오.

7-9. 다음의 경우에 대하여 쉘정렬의 실행시간을 결정하시오.

ㄱ. 정렬된 입력

*ㄴ. 거꾸로 정돈된 입력

7-10. 프로그람 7-2에서 코드화된 쉘정렬루틴에 대한 아래와 같은 수정가운데서 어느것이 최악의 경우의 실행시간에 영향을 주는가.

ㄱ. 행 2앞에서 gap가 짝수이라면 gap로부터 하나를 덜으시오.

ㄴ. 행 2앞에서 gap가 짝수라면 gap에 하나를 더하시오.

7-11. 입력 142, 143, 123, 65, 403, 829, 532, 434, 111, 242, 811, 102에 대하여 더미정렬하는 방법을 보여 주시오.

7-12. 미리 정렬된 입력에 대해 더미정렬의 실행시간은 얼마인가?

***7-13.** 더미정렬에서 어떤 잎사귀에 가도록하는 매 percolateDown에 작용하는 입력들이 있다는것을 보여 주시오(암시: 뒤방향으로 작업하시오.)

7-14. 더미정렬을 다시 작성하되 그것은 오직 범위 low로부터 high내에 있는 항목들만 정렬하도록 하시오. 이것들은 부차적인 파라메터로써 넘겨 진다.

7-15. 병합정렬을 리용하여 3, 1, 4, 1, 5, 9. 2, 6을 정렬하시오.

7-16. 재귀를 리용하지 않고 병합정렬을 어떻게 실행하는가?

7-17. 다음과 같은 입력에 대해 병합정렬의 실행시간을 결정하시오.

ㄱ. 정렬된 입력

ㄴ. 거꾸로 정렬된 입력

ㄷ. 우연적인 입력

7-18. 병합정렬에 대한 해석에서 상수들은 고려하지 않았다. 병합정렬에 의해서 최악의 경우에 리용된 비교개수는 $N\lceil \log N \rceil - 2^{\lceil \log N \rceil} + 1$이라는것을 증명하시오.

7-19. 3개중 중간분할과 3개의 차단을 가진 고속정렬을 리용하여 3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5를 정렬하시오.

7-20. 이 장에 있는 고속정렬의 실행을 리용하여 다음과 같은 입력에 대해 고속정렬의 실행시간을 결정하시오.

ㄱ. 정렬된 입력

ㄴ. 거꾸로 정렬된 입력

ㄷ. 우연적인 입력

7-21. 기준값요소가 다음과 같이 선택될 때 련습문제 7-20을 다시 푸시오.

ㄱ. 첫번째 요소

ㄴ. 처음의 첫 두개의 같지 않는 요소들중에서 더 큰것

ㄷ. 우연요소

*ㄹ. 모임안에 있는 모든 요소들의 평균값

7-22. ㄱ. 이 장에 있는 고속정렬의 실행에 대해 모든 열쇠가 같을 때 실행시간은 얼마인가?

ㄴ. 분할방법을 다음과 같이 변화시킨다고 가정하자. 즉 기준값요소와 같은 열쇠를 가진 요소를 만날 때 *i*도 *j*도 정지하지 않는다. 고속정렬이 작업한다는것을 담보하는 코드내에 어떤 상태를 만들 필요가 있으며 모든 열쇠들이 같을 때 실행시간은 얼마인가?

ㄷ. 분할방법을 다음과 같이 변화시킨다고 가정하자. 즉 *i*는 주목하는 요소와 같은 열쇠를 가진 요소에서 정지는 되지만 *j*는 이와 류사한 경우에 정지하지 않는다. 고속정렬이 작업한다는것을 담보하는 코드내에 어떤 상태를 만들 필요가 있으며 모든 키들이 같을 때 고속정렬의 실행시간은 얼마인가?

7-23. 기준값요소로서 배렬의 중간위치에 있는 요소를 택한다고 하자. 이것은 고속정렬이 두제곱시간을 요구한다는것을 믿지 않게 하는가?

7-24. 3개의 중간분할과 3개의 차단을 리용하는 고속정렬에 대해 가능한 정도 나쁜 20개 요소들의 교환을 진행하시오.

7-25. 이 책에 있는 고속정렬은 2개의 호출을 리용한다. 다음과 같이 그 호출들중의 하나를 제거하시오.

ㄱ. 두번째 재귀호출이 고속정렬에 무조건 마지막행이 되도록 코드를 작성하시오. 이것은 if와 else를 반전하고 삽입정렬에 대한 호출후에 돌려 주게 함으로써 수행하시오.

ㄴ. While순환을 쓰고 left를 선택함으로써 꼬리재귀를 제거하시오.

7-26. 련습문제 7-25를 계속하시오.

ㄱ. 더 작은 부분배렬이 첫 재귀호출에 의해서 처리되고 한편 더 큰 부분배렬은 두번째 재귀호출에 의해 처리되도록 검사를 하시오.

ㄴ. While순환을 쓰고 필요하다면 left 혹은 right중의 어느 하나를 택해서 꼬리재귀를 제거하시오.

ㄷ. 재귀호출의 개수는 최악의 경우에 로그적이라는것을 증명하시오.

7-27. 재귀적고속정렬은 초기에 거의 2log*N*인 구동프로그람으로부터 int파라메터, depth를 받는다고 가정하자.

ㄱ. 재귀준위가 depth에 도달했다면 재귀고속정렬을 그의 현재부분배렬상에서 heapsort를 호출하도록 수정하시오(암시: 재귀호출들을 할 때 depth를 감소시키시오. 그때 그것은 령이다. 더미정렬로 절환한다.)

ㄴ. 이 알고리듬의 최악의 경우의 실행시간은 $O(N\log N)$이라는것을 증명하시오.

ㄷ. 종종 heapsort가 어떻게 호출되여 얻어 지는가를 결정하는 경험들을 유도하시오.

ㄹ. 이 기술을 련습문제 7-25에서의 꼬리재귀제거와 합동하여 실행하시오.

ㅁ. 련습문제 7-26에 있는 기술이 왜 더 이상 필요하지 않는가를 설명하시오.

7-28. 고속정렬을 실행할 때 배렬이 중복된 요소들을 많이 포함한다면 재귀호출들을 보다 작게 하기 위해 3-방식분할(주목하는 요소보다 더 작다. 그리고 더 큰 요소들내에서)을 수행하는것이 더 좋다. 3-방식비교들을 가정하시오.

ㄱ. 오직 N-1번의 3-방식비교들을 리용하여 N -요소부분배렬에 대해 3-방식의 적당한 분할을 수행하는 알고리듬을 작성하시오. 기준값과 같은 d개의 항목이 있다면 d개의 추가적인 comparable교환을 리용하고 그우에 또 2-방식분할알고리듬을 리용한다.(암시:i와 j가 서로 서로 마주 향하여 이동할 때 아래와 같은 5개의 요소들로 이루어 진 그루빠들을 보존한다.

EQUAL SMALL UNKNOWN LARGE EQUAL

ㄴ. 우의 알고리듬을 리용할 때 오직 d개의 서로 다른 값들만 포함하는 N-요소배렬을 정렬하는것은 $O(dN)$시간을 가진다는것을 증명하시오.

7-29. 선택알고리듬을 실행하는 프로그람을 작성하시오.

7-30. 다음의 재귀를 푸시오: $T(N) = (1/N)\left[\sum_{i=0}^{N-1} T(i)\right] + cN,\ \ T(0) = 0$

7-31. 정렬알고리듬은 같은 요소들을 가진 요소들이 그것들이 입력에서와 같은 순서로 남아 있다면 인정한다고 본다. 이 장에 있는 어느 정렬알고리듬들이 안정하며 어느것이 그렇치 못한가? 왜 그런가?

7-32. $f(N)$개의 우연적으로 정돈된 요소들로부터 나온 N개 요소들의 정렬된 목록이 주어 졌다고 하자. 다음과 같이 가정할 때 전체 목록이 어떻게 정렬되는가?

ㄱ. $f(N)=O(1)$?

ㄴ. $f(N)=O(\log N)$?

ㄷ. $f(N)\,O(\sqrt{N})$?

*ㄹ. $O(N)$시간으로 정렬하려면 $f(N)$의 전체 목록에 관해 얼마나 클수 있는가?

7-33. N개 요소들의 정렬된 목록에서 요소 X를 찾는 알고리듬은 $\Omega(\log N)$의 비교를 요구한다는것을 증명하시오.

7-34. Stirling공식($N! \approx (N/e)^N \sqrt{2\pi N}$)을 리용하여 log(N!)에 대해 엄밀하게 평가하시오.

7-35. *ㄱ. N개 요소로 이루어 진 2개의 정렬된 배렬이 몇가지 방식으로 조합될수 있는가?

*ㄴ. N개 요소로 이루어 진 2개의 정렬된 목록을 조합하는데 요구되는 비교의 개수에 관한 비범용적인 아래한계를 주시오.

7-36. 6개 수들의 정렬에 대해 다음의 알고리듬을 고찰하시오.

- 알고리듬 *A*를 리용하여 첫 3개수들을 정렬
- 알고리듬 *B*를 리용하여 두번째부터 3개 수들을 정렬
- 알고리듬 C를 리용하여 2개의 정렬된 그루빠를 조합

알고리듬 *A, B, C*의 선택에 무관계하게 이 알고리듬이 부분최량이라는것을 보여 주시오.

7-37. *N*개의 분수들을 정렬하는 선형시간알고리듬을 작성하시오. 이때 매 분수의 분자와 분모들은 *N*과 1사이의 옹근수이다.

7-38. *A*와 *B*는 둘다 정렬되였고 *N*개의 요소를 포함한다고 가정하자. $A \cup B$의 가운데점을 찾는 $O(\log N)$알고리듬을 작성하시오.

7-39. 2개의 서로 다른 열쇠(true와 false)만 포함하는 *N*개 요소들의 배렬이 있다고 가정하자. 모든 false요소들은 true요소들보다 우위를 차지하도록 목록을 재정돈하는 $O(N)$알고리듬을 작성하시오. 오직 상수여분공간만 리용한다.

7-40. 3개의 서로 다른 열쇠(true, false, maybe)들을 포함하는 *N*개 요소들의 배렬이 있다고 하자. 모든 false요소들이 Maybe요소들보다 앞서도록 목록을 재정돈하는 $O(N)$알고리듬을 작성하시오. 이때 maybe요소들은 번갈아 true요소들보다 앞선다. 오직 상수여분공간만을 리용한다.

7-41. ㄱ. 4개 요소들을 정렬하는 임의의 비교에 기초한 알고리듬은 5번의 비교가 요구된다는것을 증명하시오.

ㄴ. 5번의 비교로 4개의 요소들을 정렬하는 알고리듬을 작성하시오.

7-42. ㄱ. 임의의 비교에 기초한 알고리듬을 리용할 때 5개의 요소들을 비교하는데 7번의 비교가 요구된다는것을 증명하시오.

*ㄴ. 7번의 비교를 가진 5개 요소들의 정렬알고리듬을 작성하시오.

7-43. 쉘정렬의 능률적인 변종을 쓰고 다음의 증분렬을 리용할 때 성능들을 비교하시오.

ㄱ. 쉘정렬의 본래렬

ㄴ. 히바드의 증분

ㄷ. knuth의 증분: $h_i = \frac{1}{2}(3^i + 1)$

ㄹ. Gomet의 증분들: $h_t = \left\lfloor \frac{N}{2.2} \right\rfloor, h_k = \left\lfloor \frac{h_{k+1}}{2.2} \right\rfloor$ (h_2=2이면 h_1=1이다.)

ㅁ. Sedgewick의 증분

7-44. 고속정렬의 최량화된 변종을 실행하고 다음의 조합들을 시험하시오.

ㄱ. 기준값요소: 첫째 요소, 중간요소, 우연요소, 3개의 중간, 5개의 중간

ㄴ. 0~20까지의 차단값

7-45. 자모순으로 배렬된 2개의 파일을 읽어서 그것들을 함께 병합하여 자모순으로

배렬된 세번째 파일을 만드는 루틴을 작성하시오.

7-46. 다음과 같이 3개의 중간에 대한 루틴을 실행한다고 하자. 즉 a[left]와 a[right] 그리고 a[center]의 중간을 찾고 그것을 a[right]와 교환한다. i는 left에서부터 j는 right-1에서부터 시작하는 표준분할단계를 계속하시오. (left+1과 right-2대신에)

ㄱ. 입력은 2,3,4,…,N-1, N,1라고 하자. 이 입력에 대해 고속정렬의 이 변종의 실행시간은 얼마인가.

ㄴ. 입력은 거꾸로 정돈되여 있다고 하자. 이 입력에 대해 고속정렬의 이 변종의 실행시간은 얼마인가.

7-47. 임의의 비교에 기초한 정렬알고리듬은 평균 $\Omega(N\log N)$의 비교를 요구한다는 것을 증명하시오.

7-48. N개의 수들을 포함하는 배렬이 주어 져 있다. 이때 그의 합이 주어 진 수 k와 같은 그런 2개의 수가 있는가를 결정하기 쉽다. 실례로 입력이 8, 4, 1, 6이고 k는 10이면 대답은 yes이다. (4와 6이 있다.)어떤 수는 2번 리용되여도 된다. 이때 다음과 같이 하시오.

ㄱ. $O(N^2)$으로 이 문제를 풀기 위한 알고리듬을 작성하시오.

ㄴ. $O(N\log N)$으로 이 문제를 풀기 위한 알고리듬을 작성하시오.
(암시: 항목들을 먼저 정렬하고 그후 선형시간으로 문제를 풀수 있다.)

ㄷ. 둘다 해답을 코드화하고 그 알고리듬들의 실행시간들을 비교하시오.

7-49. 4개의 수들에 대해 련습문제 7-48을 반복하시오.
$O(N^2\log N)$인 알고리듬을 설계하는것을 시도하시오.
(암시:두개의 요소들의 모든 가능한 합들을 계산하시오. 이 가능한 합들을 정렬하시오. 다음 련습문제 7-48에서처럼 진행하시오.)

7-50. 세 수들에 대해 련습문제 7-48을 반복하시오. $O(N^2)$알고리듬설계를 시도하시오.

7-51. percolateDown에 대해 다음의 방법을 고찰하시오. 마디 X에서 구멍을 가진다. 표준루틴은 X의 자식들을 비교하고 다음 자식을 X까지 이동한다. (만약 그것이 놓으려고 시도하고 있는 요소보다 더 크다면 더미(max)의 경우에)) 그것으로 하여 구멍을 밀어 낸다. (아래로 밀기한다.) 새 요소가 그 구멍내에 배치하는것이 안전할 때 정지한다. 또 다른 방법은 요소들은 우로 그리고 구멍은 가능한만큼 멀리 아래로 이동하는것이다. (이때 새 세포는 삽입될수 있는가 없는가에 대한 검사는 하지 않는다.) 이것은 새 세포를 잎에 배치하여 더미순서를 위반한다. 더미순서를 고착하기 위해 표준방식에서 새 세포를 우로 려과한다. 이런 사실을 포함하는 루틴을 작성하시오. 그리고 실행시간을 더미정렬의 표준실행과 비교하시오.

7-52. 오직 두개의 테프들만 리용하여 큰 파일을 정렬하는 알고리듬을 작성하시오.

7-53. ㄱ. 더미의 개수에 관해 아래한계 $N!/2^{2N}$은 buildHeap가 많아서 2N번의 비교들을 리용한다는 사실에 의해 암시된다는것을 보여 주시오.

ㄴ. 이 한계를 확장하기 위한 스털링의 공식을 리용하시오.

7-54. 프로그람 7-11에서 pointer클라스에 대해 0-파라메터설치자가 필요한가.

7-55. Null지적자를 검사하기 위해 operator*라는 성원함수를 초과적재함으로써 smart지적자클라스 pointer를 더 엄밀하게 하시오. 만약 시도가 Null지적자를 비참조하게 한다면 오유통보를 인쇄하고 만약 그렇지 않다면 *pointer를 돌려주시오.

7-56. 제7장 제8절에서 서술된 in-situ교환해석은 매개의 길이가 L인 순환들의 평균개수를 보여 줄것을 요구한다. 일상적으로 N은 정렬되는 요소들의 개수이다. p는 임의의 위치라고 하자.

ㄱ. 길이가 1인 순환때에 p가 있을 확률은 1/N이라는것을 보여 주시오.

ㄴ. 길이가 2인 순환때에 p가 있을 확률은 1/N이라는것을 보여 주시오.

ㄷ. 길이가 L인 순환때에 p가 있을 확률은 1/N이라는것을 보여 주시오.

ㄹ. ㄷ에 기초하여 길이가 L인 순환들의 기대되는 개수는 1/L이라는것을 추론하시오.(암시:매 요소는 길이가 L인 순환들의 개수에 1/L로 기여한다. 그런데 단순한 더하기는 순환들을 지나치게 계수한다.)

ㅁ. comparable복사들의 평균개수는 N-2+H_N에 의해 주어 진다는것을 보여 주시오.

참고문헌

knuth의 책 [12]는 정렬에 대해 알기 쉽게 쓴 참고서이다. Gonnet와 Baeza-Yates[5]는 더 최근의 일부 결과들과 함께 도서목록을 가지고 있다.

쉘정렬의 상세한 초기론문은 [24]이다. Hibbard[6]의 론문은 증분 2^k-1리용을 암시하였고 교환들을 피함으로써 코드를 단단히 고정시켰다. 정리 7-4는 [15]에 있다. Pratt의 아래단계 (이것은 이 책에서 암시된것보다 더 복잡한 방법을 리용한다.)는 [17]에서 찾아 볼수 있다. 개선된 증분렬들과 웃한계들은 [10],[23],[26]에 있다. 아래한계들의 일치는 [27]에서 보여 준다. 최근 결과는 어떤 증분렬도 $O(N\log N)$인 최악의 경우의 시간을 주지 않는다는것을 보여 준다[16]. 쉘정렬을 위한 평균실행시간은 여전히 미해결이다. Yao[29]는 3-증분경우에 대한 극히 복잡한 해석을 수행하였다. 결과는 더 많은 증분들로 확장되여야 하지만 최근에 약간 개선되였다[11]. 여러가지 증분렬들에 대한 시험은 [25]에서 나타난다.

더미정렬은 williams에 의해 발명되였다[28]. Floyd[2]는 더미만들기에 대한 선행시간알고리듬을 주었다. 정리 7-5는 [18]에 있다.

병합정렬에 대한 정확한 평균경우해석은 [4]에 서술했다. 여분공간이 없이 선행시간으로 조합을 실행하는 알고리듬은 [9]에 서술되였다.

고속정렬은 Hoare[7]에 있다. 이 론문은 기본알고리듬을 해석하고 대부분의 개선들을 서술하고 선택알고리듬들을 포함한다. 상세한 해석과 경험적인 연구는 Sedgewick의 학위론문 [22]의 론제였다. 대부분의 중요한 결과들은 론문 [19],[20],[21]에서 나타난다. [1]은 일부 추가적인 개선을 가진 상세한 C실행을 주며 unix의 qsort서고루틴의 많은 실행들은 2차성을 쉽게 이끌어 낸다는것을 지적한다. 련습문제 7-27은 [14]에 있다.

결정나무들과 정렬최량화는 Ford와 Jchnson[3]에서 설명하였다. 이 론문은 또한 비교들의 개수에 관하여 아래한계들을 만나는 알고리듬을 준다(그러나 기타 연산들은 아니다.). 이 알고리듬은 Manacher[13]에 의해 약간 부분최량이 되도록 보여 주었다.

외부정렬은 [12]에서 상세히 보여 주었다. 안정한 정렬(stable sorting)(련습문제 7-31에서 서술한) Horvath[8]에 의해 주소화되였다.

1. J. L. Bentley and M. D. McElroy, "Engineering a Sort Function," *Software---Practice and Experience,* 23 (1993), 1249-1265.
2. R. W. Floyd, "Algorithm 245: Treesort 3," *Communications of the* ACM, 7 (1964), 701.
3. L. R. Ford and S. M. Johnson, "A Tournament Problem," *American Mathematics Monthly,* 66 (1959), 387-389.
4. M. Golin and R. Sedgewick, "Exact Analysis of Mergesort," *Fourth SIAM Conference on Discrete Mathematics,* 1988.
5. G. H. Gonnet and R. Baeza-Yates, *Handbook of Algorithms and Data Structures,* 2nd ed., Addison-Wesley, Reading, Mass., 1991.
6. T. H. Hibbard, "An Empirical Study of Minimal Storage Sorting," *Communications of the ACM,* 6 (1963), 206-213.
7. C. A. R. Hoare, "Quicksort," *Computer Journal,* 5 (1962), 10-15.
8. E. C. Horvarh, "Stable Sorting in Asymptotically Optimal Time and Extra Space," *Journal of the ACM,* 25 (1978), 177-199.
9. B. Huang and M. Langston, "Practical In-place Merging," *Communications of the* ACM, 31(1988), 348-352.
10. J. Incerpi and R. Sedgewick, "Improved Upper Bounds on shellsort," *Journal of Computer and System Sciences,* 31 (1985), 210-224.
11. S. Janson and D. E. Knuth, "shellsort with Three Increments," *Random Structures and Algorithms,* 10 (1997), 125-142.
12. D. E. Knuth, *The Art of Computer Programming. Volume 3: Sorting and Searching,* 2d ed.,

Addison-Wesley, Reading, Mass., 1998.
13. G. K. Manacher, "The Ford-Johnson Sorting Algorithm Is Not Optimal," *Journal of the ACM,* 26 (1979), 441-456.
14. D. R. Musser, "Introspective Sorting and Selection Algorithms," *Software—Practice and Experience,* 27 (1997), 983-993.
15. Papernov and G. V. Stasevich, "A Method of Information Sorting in Computer Memories," *Problems of Information Transmission,* 1 (1965), 63-75.
16. C. G. Plaxton, B. Poonen, and T. Suel, "Improved Lower Bounds for shellsort," Proceedings of the Thirty-third Annual Symposium on the Foundations of Computer Science (1992), 226-235.
17. V. R. Pratt, *Shellsort and .Sorting Networks,* Garland Publishing, New York, 1979. (Originally presented as the author's Ph.D. thesis, Stanford University, 1971.)
18. R. Schaffer and R. Sedgewick, "The Analysis of Heapsort," *journal of Algorithms,* 14 (1993), 76-100.
19. R. Sedgewick, "Quicksort with Equal Keys," *SIAM journal on Computing,* 6 (1977), 240-267.
20. R. Sedgewick, "The Analysis of Quicksort Programs," *Acta Informatica,* 7 (1977), 27-355.
21. R. Sedgewick, "Implementing Quicksort Programs," *Communications of the ACM,* 21 (1978), 847-857.
22. R. Sedgewick, *Quicksort,* Garland Publishing, New York, 1978. (Originally presented as the author's Ph.D. thesis, Stanford University, 1975.)
23. R. Sedgewick, "A New Upper Bound for Shellsort," *journal of Algorithms, 7* (1986), 159-173.
24. D. L. Shell, "A High-Speed Sorting Procedure," *Communications of the ACM,* 2 (1959), 30-32.
25. M. A. Weiss, "'Empirical Results on the Running Time of Shellsort," *Computer journal,* 34 (1991), 88-91.
26. M. A. Weiss and R. Sedgewick, "More on Shellsort Increment Sequences," *Information Processing Letters,* 34 (1990), 267-270.
27. M. A. Weiss and R. Sedgewick, "Tight Lower Bounds for Shellsort," *journal of Algorithms,* 11 (1990), 242-251.
28. J. W. J. Williams, "Algorithm 232: Heapsort," *Communications of the ACM, 7* (1964), 347-348.
29. A.C.Yao, "An Analysis of *(h,k,* 1) Shellsort," *Journal of Algorithms,* 1(1980), 14-50.

제8장. 분리모임ADT

이 장에서는 **등가문제**를 풀기 위한 능률적인 자료구조를 서술한다. 이 자료구조는 쉽게 실현할수 있다. 매 루틴은 적은 수의 코드행들만 요구하며 단순한 배렬이 리용될수 있다. 또한 연산당 상수평균시간을 요구하므로 실현이 아주 빠르다. 이 자료구조는 리론적측면에서도 매우 흥미 있다. 그것은 그의 분석이 극히 어렵기때문인데 최악의 경우의 함수형태는 아직까지 본적이 없다. 이 장에서는 분리모임ADT에 대하여 다음과 같은 문제들을 고찰한다.

- 최소한의 코드작성으로 그것을 실현하는 방법
- 두가지 간단한 고찰을 리용하여 그의 속도를 크게 증가시키는것
- 빠른 실현에 대한 실행시간분석
- 분리모임에 대한 간단한 응용

제1절. 등가관계

요소들의 모든 쌍 (a, b), $a, b \in S$에 대하여 $a\ R\ b$가 참이든지 거짓이라면 어떠한 모임 S에 대하여 **관계**(*relation*) R가 정의된다. $a\ R\ b$가 참이면 a는 b와 관계된다고 한다.

등가관계(*equivalence relation*)는 다음의 세가지 속성을 만족하는 관계 R이다.

① (**반사성**(*reflexive*)) $a\ R\ a$ ($a \in S$)

② (**대칭성**(*symmetric*)) $a\ R\ b$ (오직 $b\ R\ a$일 때만)

③ (**이동성**(*transitive*)) $a\ R\ b$와 $b\ R\ c$는 $a\ R\ c$를 의미한다.

이에 대한 여러가지 실례들을 고찰하자.

≦관계는 등가관계가 아니다. 비록 $a \leqq a$로서 반사성을 만족시키고 $a \leqq b$, $b \leqq c$이면 $a \leqq c$로서 이동성도 만족시키지만 $a \leqq b$는 $b \leqq a$가 아니므로 대칭성을 만족시키지 않기때문이다.

전기적**련결성**(*Electrical Connectivity*)은 등가관계이다(여기서 모든 련결은 전기줄들로 되여 있다.). 이 관계는 명백히 어떤 성분이 자신과 련결되기때문에 반사적이다. a가 b에 전기적으로 련결되였다면 b는 a에 전기적으로 련결되여야 하며 따라서 관계적으로 대칭이다. 마지막으로 a가 b에 련결되고 b가 c에 련결되면 a는 c와 련결된다. 따라서 전기적련결성은 등가관계이다.

같은 나라에 있는 두 도시는 서로 관계가 있다. 이것이 **등가관계**라는것은 쉽게 알수 있다. 도로를 걸어서 a에서 b로 려행하는것이 가능하다면 도시 a는 b와 관계가 있다고 한다. 이 관계는 모든 도로들이 쌍방향이라면 등가관계로 된다.

제2절. 동적등가문제

등가관계 ～가 주어 질 때 본질적인 문제는 임의의 a와 b에 대해 a～b인가를 결정하는것이다. 만일 관계가 2차원론리변수배렬에 기억된다면 물론 이것은 상수시간에 처리될수 있다. 문제는 관계가 보통 명백하게가 아니라 암시적으로 정의된다는것이다.

실례로 등가관계가 5개 요소의 모임 $\{a_1, a_2, a_3, a_4, a_5\}$에 대하여 정의된다고 하자. 그러면 25개의 요소쌍들이 있게 되는데 그 매개는 관계가 있거나 혹은 없다. 그러나 정보 a_1～a_2, a_3～a_4, a_5～a_1, a_4～a_2는 모든 쌍들이 관계가 있다는것을 암시한다. 이것을 빨리 추리할수 있다면 좋을것이다.

요소 $a \in S$의 **등가클라스**(*equivalence class*)는 a와 관계되는 모든 요소들을 포함하는 S의 부분모임이다. 등가클라스들은 S의 일부로 이루어 진다. 즉 S의 매 성원은 명백히 하나의 등가클라스에 보인다. a～b인가를 결정하기 위해서는 a와 b가 같은 등가클라스에 속하는가 속하지 않는가 하는것만 검사하면 된다. 이것은 **등가문제**를 풀기 위한 전략을 준다.

입력은 초기에 매개 모임이 한개 요소를 가진 N개 모임들의 집합이다. 이 초기표현에서는 모든 관계들이 거짓으로 된다(반사관계들은 제외). 매 모임은 $S_i \cap S_j = \phi$이 되도록 서로 다른 요소를 가진다. 즉 **분리모임**(*disjoint*)을 만든다.

여기에서 2개의 연산이 가능하다. 첫번째 연산 find는 주어 진 요소를 포함하는 모임(즉 등가클라스)의 이름을 돌려 준다. 두번째 연산은 관계들을 추가한다. 관계 a～b를 추가하려면 먼저 a와 b가 이미 관계가 있는가를 본다. 이것은 a와 b에 대하여 find를 실행하고 그것이 같은 등가클라스내에 있는가를 검사함으로써 수행된다. 만일 그것들이 같은 클라스내에 없다면 union연산을 적용한다.[23] 이 연산은 a와 b를 포함하는 2개의 등가클라스들을 새로운 등가클라스로 병합한다. 모임의 견지에서 ∪의 결과는 처음의 모임들을 깨뜨리고 모든 모임들의 분리성을 보존하면서 새 모임 $S_k = S_i \cup S_j$를 만드는것이다. 이 리유로 이것을 수행하는 알고리듬을 흔히 분리모임 **union/find알고리듬**(*union/find algorithm*)이라고 한다.

이 알고리듬은 **동적**(*dynamic*)이다. 그것은 알고리듬실행중에 모임들이 union연산을 통하여 달라 질수 있기때문이다. 알고리듬은 또한 직결연산을 해야 한다. 즉 find연산이 실행될 때 다음 처리를 계속하기전에 대답을 주어야 한다. 또 다른 가능성은 비직결알고리듬이다. 그러한 알고리듬은 union과 find들의 전체 서렬을 알게 한다. 그것이 매 find를 위하여 제공하는 결과는 find에 이를 때까지 수행되는 모든 union들과 모순이 없어야 하지만 이 알고리듬은 모든 질문들을 다 안후에야 그에 대한 대답을 줄수 있다. 그 차이는

---

[23] Union 은 C++의 예약어이다. Union/find 알고리듬서술의 전반에서 이것을 리용하지만 코드를 작성할 때 그 성원함수를 unionSets 라고 명명한다.

필답시험과 구답시험을 실시하는것과 류사하다. 필답시험은 일반적으로 비직결이라고 할수 있다. 즉 시간이 끝나기전에 대답이 주어 져야 한다. 그러나 구답시험은 직결이다. 왜냐면 다음의 질문이 진행되기전에 현재질문에 대답해야 하기때문이다.

요소들의 관계값들을 비교할 때 아무런 연산도 수행하지 않지만 그 위치에 대한 정보는 꼭 필요하다. 이러한 리유로 모든 요소들은 0~N-1까지 순차적으로 번호를 붙이며 그 번호는 일부 하쉬체계들에 의해 쉽게 결정할수 있다고 가정한다. 결국 초기에 0부터 N-1까지의 i에 대하여 $S_i=\{i\}$이다.[24]

두번째 고찰은 find에 의해 되돌려 지는 모임의 이름이 실제로 매우 우연적이라는것이다. 문제는 find(a)==find(b)가 a와 b가 같은 모임에 속할 때만 참으로 된다는것이다.

이 연산들은 많은 그라프리론문제들과 등가(또는 형)선언을 처리하는 번역기들에서 중요하다. 이에 대한 응용은 후에 고찰한다.

이 문제를 푸는데 두가지 전략이 있다. 하나는 find명령이 최악의 경우 상수시간에 실행될수 있다는것을 보증하는것이며 다른 하나는 union명령이 최악의 경우 상수시간에 실행될수 있다는것을 보증하는것이다. 최근에는 두 방법 다 최악의 경우 상수시간에 동시에 수행될수 없다는것이 밝혀 졌다.

이제 첫번째 방법을 간단히 보기로 하자. find연산을 빨리 수행하도록 하기 위하여 매 요소에 대한 등가클라스의 이름을 배렬로 보존한다. 그러면 find연산은 단순히 $O(1)$의 탐색일뿐이다. union(a, b)를 수행하려고 한다고 하자. a는 등가클라스 i에, b는 등가클라스 j에 속한다고 가정하고 모든 i들을 j로 변화시키면서 배렬을 아래로 조사한다. 유감스럽게도 이 조사는 $\Theta(N)$을 가진다. 따라서 N-1개의 union연산서렬(그때 모든 요소들이 한 모임에 있기때문에 최대값으로 되는)들은 $\Theta(N^2)$시간을 가진다. $\Omega(N^2)$의 find연산을 가진다면 이 성능은 좋다. 그것은 알고리듬실행의 전과정에 총 실행시간이 매 union연산이나 find연산에 대해 $O(1)$로 되기때문이다. 만일 find연산들이 더 적다면 이 한계는 불만족하다.

한가지 방법은 같은 등가클라스에 속하는 모든 요소들을 련결목록에 보존하는것이다. 이렇게 하면 갱신할 때 배렬전체를 탐색하지 않아도 되기때문에 시간을 절약할수 있다. 이것은 알고리듬실행의 전과정에 여전히 $\Theta(N^2)$의 등가클라스갱신이 있을수 있으므로 자기자체로 점근실행시간을 감소시키지 못한다.

만일 매 등가클라스의 크기의 변화를 기억하고 있고 union을 수행할 때 보다 작은 등가클라스의 이름을 더 큰것으로 변화시킨다면 N-1개의 병합에 걸리는 총체적인 시간은 O(Nlog$N$)이다. 그 리유는 클라스가 변화될 때마다 새로운 등가클라스가 적어도 본래의 두배로 되므로 매 요소는 많아서 log$N$번 변화된 등가클라스를 가질수 있기때문이다. 이 전략을 리용함으로써 M개의 find연산과 N-1개까지의 union연산들의 서렬은 기껏해서 O(M+Nlog$N$)시간을 가진다.

[24] 이것은 배렬의 첨수가 0에서부터 시작한다는것을 말한다.

이 장의 나머지 부분에서는 union연산들은 쉽지만 find연산들은 힘들게 하는 /find문제에 대한 해결을 고찰한다. 그렇다 해도 최대 M개의 find연산들과 N-1개까지의 union연산들의 서렬에 대한 실행시간은 O(M+N)보다 약간 많아 질뿐이다.

제3절. 기본자료구조

문제는 find연산이 어떤 정확한 이름을 되돌리는것을 요구하지 않으며 2개의 요소들에 대한 find연산은 그것들이 같은 모임에 있을 때에만 같은 결과를 돌려 준다는것을 상기해야 한다. 한가지 방법은 나무에 속한 매 요소는 같은 뿌리를 가진다는데로부터 나무를 리용하여 매개 모임을 표현하는것이다. 따라서 뿌리는 모임의 이름으로 리용될수 있다. 매 모임을 나무로 표현하자(나무들의 집합을 수림이라고 한다.). 초기에 매 모임은 한개 요소를 포함한다. 여기에서 리용하는 나무는 반드시 2진나무일 필요는 없지만 그의 표현은 쉽다. 그것은 요구되는 정보는 오직 부모에 대한 지적자뿐이기때문이다. 모임의 이름은 뿌리에 있는 매듭에 의해 주어 진다. 부모의 이름만 요구된다는데로부터 이 나무는 배렬에 암시적으로 기억된다고 가정할수 있다. 즉 배렬에서 매 항목 s[i]는 요소 i의 부모를 표현한다. i가 뿌리이면 s[i]=−1이다. 그림 8-1의 수림에서 s[i]=−1(0≦i<8)이다. 2진더미들에 의해서 배렬이 리용되고 있다는것을 리해할수 있도록 나무들을 명백하게 묘사한다. 그림 8-1은 명시적인 표현을 보여 준다. 편리상 뿌리의 부모련결은 수직으로 묘사한다.

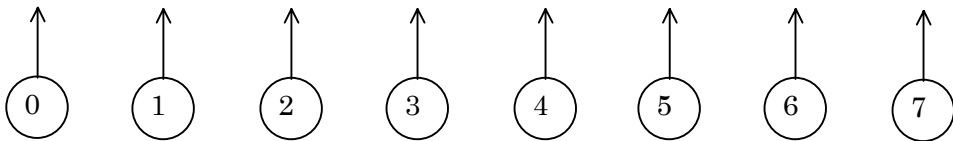



그림 8-1. 초기에 서로 다른 모임에 속하는 8개의 요소들

두개의 모임을 가지고 union을 수행하기 위하여 2개의 나무를 다음과 같이 병합한다. 즉 한 나무의 뿌리의 부모련결을 다른 나무의 뿌리매듭에 련결한다. 이 연산이 상수시간을 가진다는것은 명백하다. 그림 8-2～8-4는 union(4,5), union(6,7), union(4,6)의 매개 연산들이 수행된 다음의 수림을 표현한다. 여기서 union(x,y)이후의 새로운 뿌리가 x라는 관계를 리용하였다. 마지막수림에 대한 암시적인 표현을 그림 8-5에 보여 주었다.

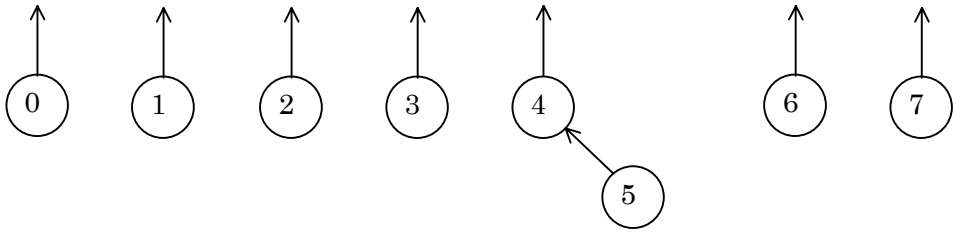



그림 8-2. union(4,5)이 실행된후

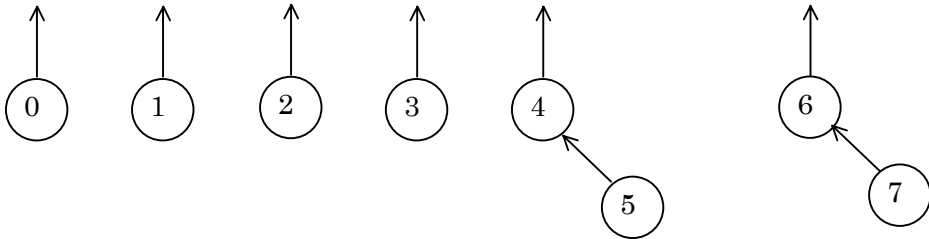



그림 8-3. union(6,7)이 실행된후

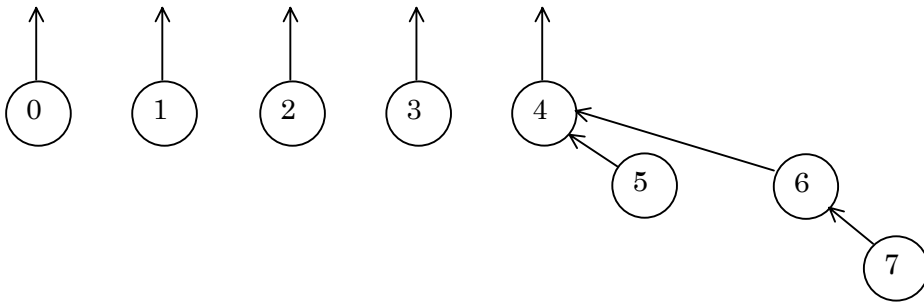



그림 8-4. union(4,6)이 실행된후

-1	-1	-1	-1	-1	4	4	6
0	1	2	3	4	5	6	7

그림 8-5. 우의 나무의 암시적표현

요소 x에 대한 find(x)는 x를 포함하는 나무의 뿌리를 돌려 주는 방법으로 수행된다. 이 연산의 수행시간은 x를 표현하는 매듭의 깊이에 비례한다(물론 x를 표현하는 매듭을

상수적인 시간에 찾을수 있다는 조건에서). 우의 전략을 리용하면 깊이가 N-1인 나무를 생성하는것이 가능하다. 따라서 find의 최악의 경우의 실행시간은 $O(N)$이다. 일반적으로 실행시간은 M개의 혼합된 명령들의 서렬에 대해 계산된다. 이 경우에 M개의 련속적인 연산은 최악의 경우에 O(MN)시간을 가진다.

프로그람 8-1~8-4의 코드는 오유검사가 이미 진행되였다고 가정하고 **기본알고리듬의 실현**을 표현한다. 이 루틴에서 union들은 나무들의 뿌리에 대하여 수행된다. 보통 그 연산은 union이 임의의 두개 요소들을 넘겨 받고 뿌리를 결정하기 위하여 두개의 find를 실행하는것으로 수행된다. 앞에서 본 자료구조들에서 find는 늘 접근자였으며 따라서 const성원함수였다. 제8장 제5절에서는 보다 효과적인 변종을 설명한다. 두가지가 다 동시에 지원될수 있다. 조종대상이 수정불가능하지 않는 한에는 변종이 호출된다.

```
class DisjSets
{
    public:
        explicit DisjSets( int numElements );

        int find( int x ) const;
        int find( int x );
        void unionSets( int root1, int root2 );

    Private:
        vector<int> s;
};
```

프로그람 8-1. 분리모임클라스대면부

```
/**
 *Construct the disjoint sets object.
 *numElements is the initial number of disjoint sets.
 */
DisjSets::DisjSets( int numElements ) : s(numElements )
{
    for( int i = 0; i < s.size( ); i++ )
        s[ i ] = -1;
}
```

프로그람 8-2. 분리모임초기화루틴

```
/**
  * Union two disjoint sets.
  * For simplicity, we assume root1 and root2 are distinct
  * and represent set names.
  * root1 is the root of set 1.
  * root2 is the root of set 2.
  */
void DisjSets::unionSets( int root1, int root2 )
{
        s[ root2 ] = root1;
}
```

프로그람 8-3. union(제일 좋은 방식은 아니다.)

```
/**
  * Perform a find.
  * Error checks omitted again for simplicity.
  * Return the set containing x.
  */
int DisjSets::find( int x ) const
{
    if( s[ x ] < 0 )
        return x;
    Else
        return find( s[ x ] );
}
```

프로그람 8-4. 단순한 분리모임 find알고리듬

평균경우분석은 대단히 힘들다. 최소한의 문제들은 결과가 평균을 정의하는 방법에 관계된다는것이다(union연산에 대하여). 실례로 그림 8-4에 있는 수림에는 5개의 나무가 있으므로 다음의 union에 대한 균등하게 적당한 결과들은 5・4=20개이라고 할수 있다(임의의 2개의 서로 다른 나무들이 결합될수 있기때문에). 물론 이 모형의 암시는 다음의 union이 큰 나무를 포함할 기회가 2/5만 있다는것이다. 또 다른 모형은 서로 다른 나무에 속하는 2개의 요소들사이의 union들은 모두 균등하게 알맞으며 따라서 보다 큰 나무는 보다 작은 나무에 비하여 다음의 union에 포함되는것이 더 적당하다고 말할수 있다. 우의 실례에서 {0, 1, 2, 3}의 두개 요소를 병합하는데는 6개의 방식이, {4, 5, 6, 7}의 요소를 {0, 1, 2, 3}의 요소와 병합하는데는 16개 방식이 있다는데로부터 큰 나무가 다음의

union내에 포함되는데는 8/11의 기회가 있다. 이밖에도 여러가지 모형들이 있으나 어느것이 가장 좋다고는 말할수 없다. 평균실행시간은 모형에 관계된다. 즉 $\Theta(M)$, $\Theta(M\log N)$, $\Theta(MN)$의 한계는 서로 다른 세개의 모형들에 대해서 실제로 보여 진다(비록 맨 마지막 한계가 더 현실적이라고 생각되더라도).

연산렬에 대한 2차실행시간은 일반적으로 허용할수 없다. 다행히도 이 실행시간이 생기지 않는다는것을 쉽게 보증하는 방법들이 몇가지 있다.

제4절. 적응union알고리듬

우의 union들은 둘째 나무를 첫째 나무의 부분나무로 만들어서 약간 제멋대로 수행되였다. 이것을 약간 개선하여 항상 보다 작은 나무를 보다 큰 나무의 부분나무로 되게 한다. 이때 련결매듭들은 임의의 방법으로 끊어 놓는다. 이 방법을 **크기에 의한 결합** (*union-by-size*)이라고 한다. 앞의 실례에서 세개의 union은 모두 련결매듭들이며 따라서 그것들이 크기에 의해 수행되였다고 생각할수 있다. 다음번 연산이 union(3,4)라면 그림 8-6에 있는 수림이 형성된다. 크기에 따르지 않는다면 깊이가 더 깊은 나무가 형성된다 (그림 8-7).

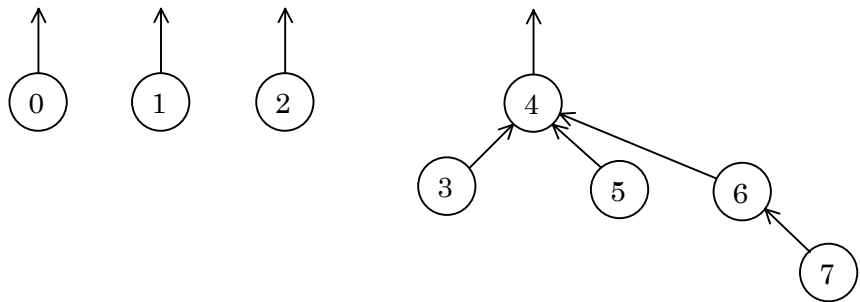



그림 8-6. 크기에 의한 union(union-by-size)의 결과

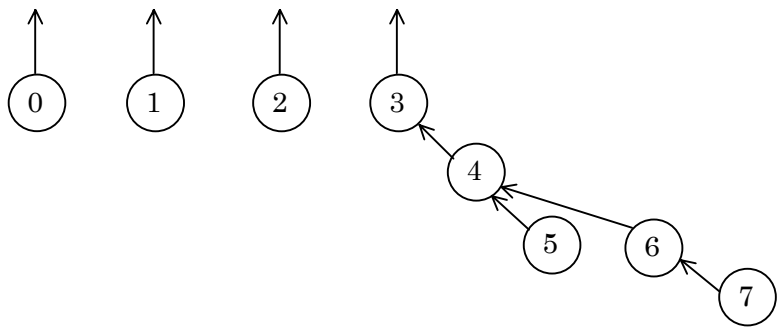



그림 8-7. 임의의 union 의 결과

union들이 크기에 의해 수행된다면 임의의 매듭의 깊이는 $\log N$이상일수 없다는것을 증명할수 있다. 이것을 보기 위해 매듭은 초기에 깊이 0에 있다는것을 주의해야 한다. union의 결과로 매듭의 깊이가 증가할 때 그것은 나무에서 적어도 2배만큼 깊은 위치에 놓여 진다. 결국 그의 깊이는 최대로 $\log N$배 증가될수 있다(제8장 제2절의 마지막에 있는 **고속find알고리듬**에서 이 인수를 리용하였다.). 이것은 find연산에 대한 실행시간이 $O(\log N)$이며 M개 연산들의 서렬은 $O(M\log N)$만큼 시간이 걸린다는것을 암시한다. 그림 8-8에 있는 나무는 16번의 union실행이후의 가능한 최악의 나무를 보여 주며 모든 union들이 같은 크기의 나무들사이에 진행되는 경우에 얻어 진다(최악의 경우 나무들은 제6장에서 설명한 2항나무들이다.).

이 전략을 실행하기 위하여 매 나무의 크기변화를 기억할 필요가 있다. 실제로 배렬을 리용하고 있으므로 매 뿌리에 대하여 그 나무의 크기만한 부의 값을 가지는 배렬항목을 가질수 있다. 따라서 나무의 배렬표현은 초기에 모두 -1이다. union을 수행할 때 크기를 검사한다. 새로운 크기는 낡은것들의 합이다. 결국 크기에 의한 union은 실행하기가 그리 힘들지 않으며 어떤 여분공간도 요구하지 않는다. 그것은 또한 평균보다 빠르다. 거의 모든 합리적인 모형들에 대하여 M개 연산들의 서렬은 크기에 의한 union이 리용된다면 $O(M)$평균시간을 요구한다는것을 보여 주었다. 그것은 어떤 union이 수행될 때 일반적으로 매우 작은(보통 한개의 요소) 모임들이 알고리듬의 전과정에 큰 모임들과 병합되기 때문이다.

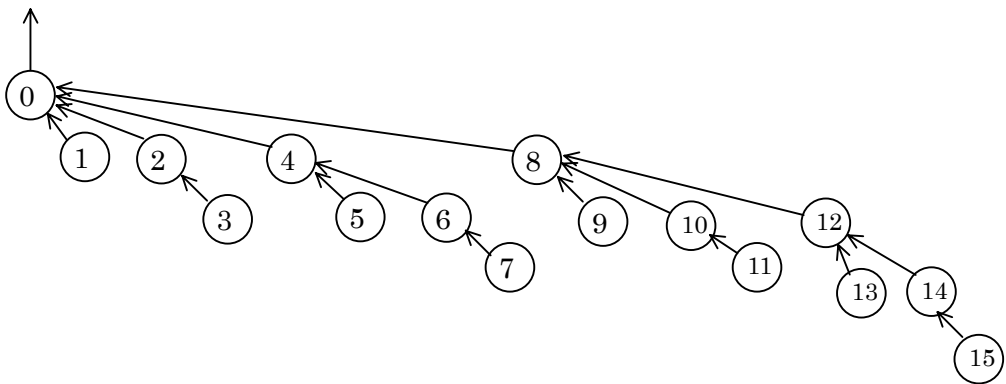
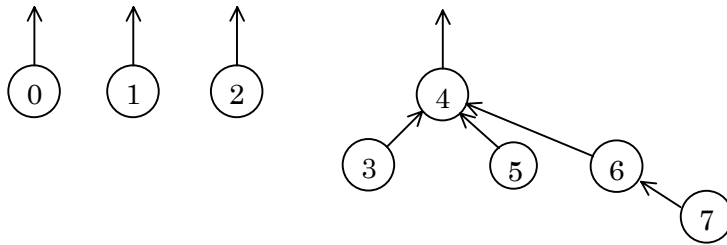



그림 8-8. N=16 에 대한 최악의 경우의 나무

모든 나무들이 최대로 $O(\log N)$의 깊이를 가진다는것을 담보하는 또 다른 안은 **높이에 의한 결합**(*union-by-height*)이다. 매 나무에 대하여 크기대신에 높이를 기억하며 얕은 나무는 보다 깊은 나무의 부분나무로 만들어서 union들을 수행한다. 이것은 쉬운 알고리듬이다. 그것은 나무의 높이가 깊이가 같은 두개의 나무를 합할 때에만 증가하기때문이다(그때 높이는 하나 증가한다.). 그러므로 높이에 의한 union은 크기에 의한 union의 약간한 수정으로 된다. 높이가 령일 때는 부수가 아니므로 실지로는 1을 덜어 낸 부수의

높이를 기억한다. 초기에 모든 항목들은 –1이다.

다음의 그림들은 크기에 의한 union과 높이에 의한 union에 대한 나무와 그의 암시적표현을 보여 준다. 프로그람 8-5에 있는 코드는 높이에 의한 union을 실현한다.

−1	−1	−1	4	−5	4	4	6
0	1	2	3	4	5	6	7

−1	−1	−1	4	−3	4	4	6
0	1	2	3	4	5	6	7

```
/**
  * Union two disjoint sets.
  * For simplicity, we assume root1 and root2 are distinct
  * and represent set names.
  * root1 is the root of set 1.
  * root2 is the root of set 2.
  */
void DisjSets::unionSets( int root1, int root2 )
{
        if( s[ root2 ] < s[ root1 ] )            // root2 is deeper
             s[ root1 ] = root2;                 // Make root2 new root
        Else
        {
             if( s[ root1 ] == s[ root2 ] )
                  s[ root1 ]--;                  // Update height if same
             s[ root2 ] = root1;                 // Make root1 new root
        }
}
```

프로그람 8-5. 높이(위수) 에 의한 union코드

제5절. 경로압축

지금까지 설명한 union/find알고리듬은 많은 경우 완전히 접수할수 있다. 그것은 매우 간단하며 *M*개 명령렬들에 대하여 대체로 선형적이다(모든 모형들에서). 그러나 O(*M*log*N*)인 최악의 경우가 상당히 쉽게 자연적으로 발생할수 있다. 실례로 모든 모임들을 대기렬에 넣고 첫 두개 모임을 대기렬에서 제거하고 그의 union을 대기렬에 넣는것을 반복한다면 최악의 경우가 생긴다. 만일 find가 union보다 훨씬 더 많다면 이 실행시간은 고속find알고리듬의 실행시간보다 더 나쁘다. 더우기 union알고리듬에 대한 개선이 더는 가능하지 않다는것은 명백하다. 이것은 union을 실행하는 방법들은 어느것이나 다 련결매듭을 마음대로 끊어 내므로 같은 최악의 경우 나무들을 산생할것이라는 고찰에 기초하고 있다. 그러므로 자료구조에 대한 전반적인 수정없이 알고리듬의 속도를 높이기 위한 유일한 방법은 find연산에 대하여 재치를 부리는것이다.

그 재치 있는 연산은 **경로압축**(*path compression*)으로 알려 져 있다. 경로압축은 find연산시에 수행되며 union을 수행하는데 리용되는 전략에는 무관계하다. 연산이 find(x)라고 하자. 이때 경로압축효과는 x에서 뿌리까지의 경로우에 있는 모든 매듭이 뿌리로 변화된 부모를 가진다는것이다. 그림 8-9는 그림 8-8의 일반적인 최악의 나무에 대하여 find(14)를 수행한후의 경로압축효과를 보여 준다.

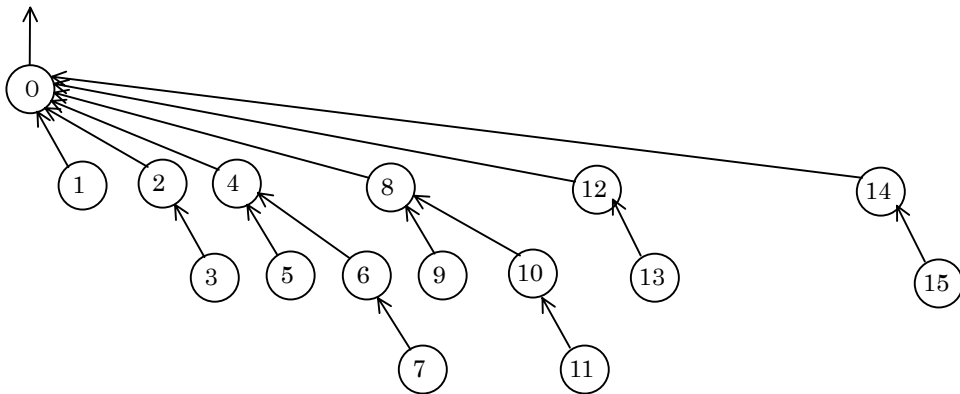



그림 8-9. 경로압축의 실례

경로압축의 효과는 여분의 두개의 련결변화로 매듭 12와 13은 뿌리에 하나 더 가까운 위치로 되고 매듭 14와 15는 둘 더 가까운 위치로 된것이다. 결국 여분의 작업으로 경로압축을 실현하면 앞으로는 이 매듭들에 대하여 빨리 호출할수 있을것이다.

프로그람 8-6에 있는 코드에서 보는바와 같이 경로압축은 기본find알고리듬에 약간한 변경을 가했을뿐이다. find루틴에 대한 변화는 s[x]가 find에 의해 되돌려 진 값과 같게

된다는것 즉 모임의 뿌리를 재귀적으로 찾은후 x의 부모련결이 그것을 참조한다는것뿐이다(find루틴이 더는 const성원함수가 아니라는 사실은 제외하고). 이것은 뿌리까지의 경로우에 있는 모든 매듭들에 대하여 재귀적으로 생기며 이렇게 하여 경로압축이 실현된다.

```
/**
 * Perform a find with path compression.
 * Error checks omitted again for simplicity.
 * Return the set containing x.
 */
int DisjSets::find( int x )
{
        if( s[ x ] < 0 )
            return x;
        Else
            return s[ x ] = find( s[ x ] );
}
```

프로그람 8-6. 경로압축을 리용한
분리모임 find 연산코드

union들이 임의로 진행될 때 경로압축은 좋은 구상이다. 왜냐하면 다량의 깊은 매듭들이 있는데 이것들은 경로압축에 의하여 뿌리에 더 가까워 지기때문이다. 이 경우에 경로압축이 진행되면 M개의 연산렬이 기껏해서 $O(M\log N)$시간을 요구한다는것이 증명되였다. 현 상황에서 평균경우동작을 결정하는것은 여전히 해결되지 않은 문제이다.

경로압축은 크기에 의한 union들과 완전히 호환될수 있으며 따라서 두 루틴은 같은 시간내에 실행될수 있다. 그자체로 크기에 의한 union을 진행하는것은 M개의 연산렬을 선형시간내에 실행할것을 기대하기때문에 경로압축에 포함된 여분의 통로가 항상 가치있는것은 아니다. 실제로 이 문제는 여전히 해결책이 없다. 그러나 후에 보게 되는것처럼 경로압축과 적응union규칙의 결합은 모든 경우에 매우 효과적인 알고리듬을 담보한다.

경로압축은 높이에 의한 union들과 전혀 호환되지 않는다. 왜냐하면 경로압축은 나무들의 높이를 변화시킬수 있기때문이다. 그것들을 능률적으로 다시 계산하는 방법은 확실한것이 없다. 결론은 그렇게 하지 말라는것이다. 매 나무에 대하여 기억된 높이는 추정된 높이(때로는 **위수**(rank)라고도 한다.)이며 위수에 의한 union이 리론적으로 크기에 의한 union만큼 능률적이라는것을 알게 된다. 더우기 높이는 크기보다 더 적게 갱신된다. 크기에 의한 union에서처럼 경로압축이 항상 가치 있겠는가 하는것은 명백치 않다. 다음 절에서 보게 되는것은 어느 한 **통합(*union*)수법**으로 경로압축이 최악의 경우의 실행시간을 대폭 줄인다는것이다.

제6절. 위수에 의한 union의 최악의 경우와 경로압축

두가지가 다 리용될 때 알고리듬은 최악의 경우에 거의 선형이다. 특히 최악의 경우에 요구되는 시간은 $\Theta(M\alpha(M,N))$($M \geqq N$인 조건에서)이다. 여기서 $\alpha(M,N)$은 **악커만함수**의 함수적인 반전이다. 이것은 다음과 같이 정의된다.[25]

$$A(1, j) = 2^{j}, \quad j \geqq 1\text{일 때}$$
$$A(i, 1) = A(i\text{-}1, 2), \quad i \geqq 2\text{일 때}$$
$$A(i, j) = A(i\text{-}1, A(i, j\text{-}1)), \quad i, j \geqq 2\text{일 때}$$

이로부터

$$\alpha(M, N) = \min\{i \geq 1 \mid A(i, \lfloor M / N \rfloor) > \log N\}$$

로 정의한다.

값들을 계산해 보면 실천적으로 $\alpha(M, N) \leqq 4$이며 실지로 중요한것은 바로 이것이다. 단일변수역악커만함수(때때로 $\log^* N$으로 쓴다.)는 $N \leqq 1$일 때까지 N에 log를 취하는 회수이다. 따라서 $\log^* 65536=4$이다. 왜냐하면 loglogloglog65536=1이기때문이다. $\log^* 2^{65536}=5$이지만 2^{65536}은 2만자리수라는것을 명심하시오. $\alpha(M,N)$은 $\log^* N$보다 균일하게 천천히 커진다. 그러나 $\alpha(M,N)$은 상수가 아니므로 실행시간은 선형이 아니다.

이 절의 나머지 부분에서 약간 빈약한 결과를 증명하게 된다. $M=\Omega(N)$개의 union/find연산들의 서렬은 총 $O(M\log^* N)$의 실행시간을 가진다는것을 알수 있다. 위수에 의한 union이 크기에 의한 union과 교체되면 같은 한계를 가진다. 이 분석은 아마 이 책에서 가장 복잡할것이며 실현이 그리 어렵지 않은 알고리듬에 대하여 진행한적이 있는 가장 복잡한 최악의 경우에 대한 해석들중의 하나일것이다.

1. union/find 알고리듬의 분석

이 절에서는 $M=\Omega(N)$인 union/find연산렬들의 실행시간에 대하여 상당히 정확한 한계를 설정한다. union과 find들은 임의의 순서로 발생하지만 union들은 위수에 의해 진행되며 find들은 경로압축으로 수행된다.

위수가 r인 매듭수에 관계되는 몇가지 보조정리들을 설정하는것으로 시작한다. 직관적으로도 위수에 의한 union의 규칙때문에 위수가 큰 매듭보다 위수가 작은 매듭이 더 많다. 특히 위수가 $\log N$인 매듭은 기껏해서 1개 있을수 있다. 이제 진행하려는것은 위수

[25] 악커만의 함수는 흔히 $j \geqq 1$에 대하여 $A(1,j)=j+1$로 정의된다. 이 함수가 빨리 증가할 때 역함수는 보다 천천히 증가한다.

가 어떤 지정된 값 r인 매듭수에 대하여 가능한껏 정확한 한계를 생성하는것이다. 위수는 오직 union이 수행될 때(그리고 두개의 나무가 같은 위수를 가질 때)에만 변하므로 경로압축을 무시하는것으로써 이 한계를 증명할수 있다.

보조정리 8-1

union명령렬을 실행할 때 위수가 r인 매듭은 자신도 포함하여 적어도 2^r개의 자손들을 가져야 한다.

증명:

귀납법. 기초 r=0은 명백히 참이다. T는 제일 적은 수의 자손을 가진 위수 r인 나무이고 X는 T의 뿌리라고 하자. X가 포함된 최후의 union이 T_1과 T_2사이에 있었다고 하자. T_1의 뿌리는 X였다고 하자. T_1이 위수 r를 가진다면 T_1은 T보다 더 적은 자손들을 가지는 높이가 r인 나무이며 이것은 T가 가장 적은 수의 자손을 가진 나무라는 가정에 모순된다. 따라서 T_1의 위수≦r-1 이고 T_2의 위수≦T_1의 위수이다. T는 위수 r를 가지며 그 위수는 오직 T_2때문에만 증가할수 있다는데로부터 T_2의 위수=r-1 이라는것이 나온다. 이때 T_1의 위수=r-1이다. 귀납법전제에 의해서 매 나무는 적어도 2^r-1개의 자손들을 가진다. 그래서 총 2^r개가 주어 지고 보조정리가 성립된다.

보조정리 8-1은 어떤 경로압축도 수행하지 않는다면 위수가 r인 임의의 매듭은 적어도 2^r개의 자손들을 가져야 한다는것을 의미한다. 물론 경로압축은 매듭으로부터 자손들을 제거할수 있으므로 이것을 변화시킬수 있다. 그러나 union들이 수행될 때 경로압축과 동등하게 위수를 리용하고 있는데 이것은 추정된 높이이다. 이 위수들은 어떤 경로압축도 없는것처럼 작용한다. 결국 위수 r인 매듭개수를 한계 지을 때 경로압축은 무시될수 있다.

따라서 다음의 보조정리는 경로압축이 있건 없건 관계없이 유효하다.

보조정리 8-2.

위수가 r인 매듭개수는 기껏해서 $N/2^r$이다.

증명:

경로압축이 없을 때 위수 r인 매 매듭은 적어도 2^r개의 매듭을 가지는 부분나무의 뿌리이다. 부분나무에서 어떤 매듭도 위수 r를 가질수 없다. 따라서 위수 r인 매듭을 가지는 모든 부분나무들은 서로 사귀지 않는다. 그러므로 기껏해서 $N/2^r$개의 서로 사귀지 않는 부분나무들이 있으며 이로부터 위수가 r인 매듭은 $N/2^r$개이다.

다음의 보조정리는 어느 정도 리해하기는 쉽지만 분석은 어렵다.

보조정리 8-3.

union/find알고리듬의 임의의 시점에서 잎으로부터 뿌리까지의 경로상에 있는 매듭들의 위수는 단조증가한다.

증명:

보조정리는 아무런 경로압축도 없다면 명백하다(실례를 보시오.). 만일 경로압축후에 어떤 매듭 *v*가 *w*의 자손이라면 명백히 *v*는 union만이 고려되였을 때 *w*의 자손이여야 한다. 이로부터 *v*의 위수는 *w*인 위수보다 작다.

지금까지 취급한 내용들을 종합하여 결론해 보자. 보조정리 8-2는 몇개의 매듭들이 위수 *r*에 지정될수 있는가를 말해 준다. 위수는 union들에 의해서만 지정되기때문에(이것은 경로압축에 대한 어떠한 구상도 가지지 않는다.) 보조정리 8-2는 union/find알고리듬의 모든 단계들과 지어 경로압축에서도 타당하다. 그림 8-10은 위수 0과 1에 많은 매듭들이 있으며 r가 클수록 위수 r인 매듭은 더 적다는것을 보여 준다.

보조정리 8-2는 임의의 위수 *r*에 대해 거기에 $N/2^r$개의 매듭들이 있을수 있다는것을 의미한다. 한계가 모든 위수 *r*에 대하여 동시에 만족되도록 하는것은 불가능하기때문에 그것은 어느 정도 여유가 있다. 보조정리 8-2가 위수 *r*인 매듭의 개수를 표현한다면 보조정리 8-3은 그의 분포(distribution)를 말해 준다. 한마디로 말하여 매듭의 위수는 잎으로부터 뿌리에 이르는 경로를 따라 가면서 정확히 증가하고 있다.

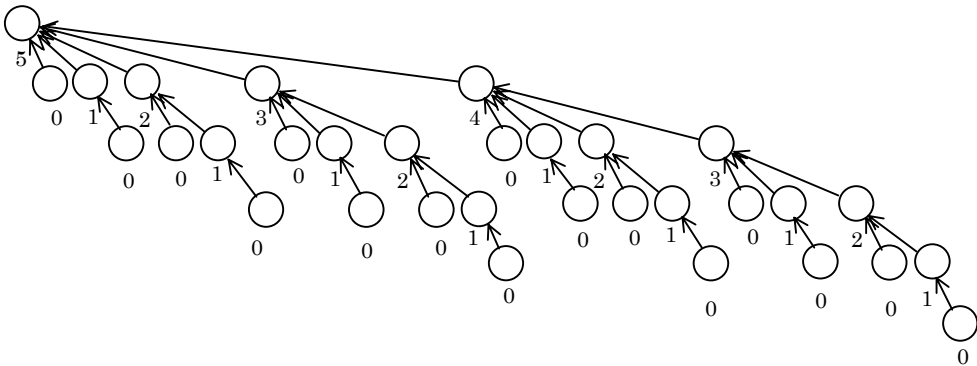



그림 8-10. 큰 분리모임나무(매듭밑에 있는 수자들은 위수이다.)

이제는 기본정리를 증명할 준비가 되였다. 기본사상은 다음과 같다. 즉 임의의 매듭 *v*에 대한 find는 *v*로부터 뿌리까지의 경로우에 있는 매듭들의 개수에 비례하여 시간이 소비된다. 그때 매 find에 대하여 *v*로부터 뿌리까지의 경로상에 있는 매 매듭들에 대한 비용을 계산한다고 하자. 비용을 계수하기 위하여 경로우에 있는 매 매듭에 가상적으로 잔돈을 예금한다. 이것은 엄밀하게 프로그람부분이 아닌 회계속임수이다. 알고리듬이 끝날

때 예금된 잔돈들을 모두 물는다. 이것이 바로 총 비용이다.

앞으로 회계속임수로서 미국과 카나다잔돈들을 둘다 예금한다. 알고리듬실행시 매 find에 대하여 일정한 개수의 미국잔돈들만 예금할수 있으며 매 매듭에는 일정한 수의 카나다잔돈만 예금할수 있다. 이 두가지를 더한것이 예금될수 있는 잔돈들의 총수에 대한 한계이다.

이제 좀 더 상세하게 회계도식을 요약하여 보자. 위수에 따라 매듭들을 나눈다. 다음 위수들을 위수그룹들로 나눈다. 매 find에 대하여 미국돈은 일반적인 공동자금으로, 카나다돈은 지정된 정점들에 예금한다.[26] 예금된 카나다돈의 총수를 계산하기 위해서 매듭마다 예금을 계산한다. 위수 r인 매듭들에 대한 예금들을 모두 합하면 위수 r에 대한 총 예금을 얻는다. 다음 그룹 g에 있는 매 위수 r에 대한 예금들을 모두 합하여 매 위수그룹 g에 대한 총 예금을 얻는다. 마지막으로 매 렬그룹 g에 대한 예금들을 모두 합하여 수림에 예금된 카나다돈의 총수를 얻는다. 이것을 공동자금에 있는 미국돈과 합하여 결과를 얻는다.

위수들을 그룹으로 분할한다. 위수 r는 그룹 $G(r)$에 속하는데 G는 후에 결정한다. 임의의 위수그룹 g안에 있는 가장 큰 위수는 $F(\mathrm{g})$이다. 여기서 $F=G^{-1}$은 G의 거꿀이다. 임의의 위수그룹 g>0안에 있는 위수들의 개수는 결국 $F(\mathrm{g})-F(\mathrm{g}-1)$이다. 확실히 $G(N)$은 가장 큰 위수그룹에 관해서 매우 여유가 많은 웃한계이다. 실례로 표 8-1과 같이 위수들을 분할한다고 하자. 이 경우에 $G(r)=\left|\sqrt{r}\right|$이다. 그룹 g에 있는 가장 큰 위수는 $F(\mathrm{g})=\mathrm{g}^2$이며 그룹 g>0은 $F(\mathrm{g}-1)+1$부터 $F(\mathrm{g})$까지(그자체도 포함)의 위수들을 포함한다는것을 고려한다. 이 공식은 위수그룹 0에 대해서는 적용하지 않으며 편리상 그룹 0은 위수가 0인 요소들만을 포함한다고 본다. 그룹들은 련속적인 위수들로 이루어 진다는것을 명심하시오.

표 8-1. 가능한 위수그룹분할

그룹	위수
0	0
1	1
2	2,3,4
3	5 ~ 9
4	10 ~ 16
i	$(i-1)^2+1$ ~ i^2

앞에서 언급된것처럼 매 뿌리가 그의 부분나무들이 얼마나 큰가를 기억하고 있는 한에는 매 union명령이 상수시간을 가진다. 따라서 union들은 이 증명이 진행되는 한 기본적으로 제한이 없다.

매 find(i)는 i로 표현되는 정점으로부터 뿌리까지의 경로상에 있는 정점들의 개수에 비례하는 시간을 가진다. 그래서 경로상에 있는 매 정점에 대하여 잔돈을 하나씩 예금한다. 그러나 이것이 전부라면 경로압축의 우점을 취하지 못하므로 거의 한계를 기대할수 없다. 그러므로 분석에서는 경로압축의 우점을 취해야 한다. 특종의 회계를 리용한다.

[26] 매듭과 정점을 구별없이 리용한다.

i를 표현하는 정점으로부터 뿌리까지의 경로상에 있는 매 정점 v에 대하여 다음의 회계들중의 하나로 한개의 잔돈을 예금한다.

① v가 뿌리이거나 v의 부모가 뿌리이거나 v의 부모가 v와 다른 위수그룹에 있다면 이 규칙으로 예금한다. 즉 미국잔돈을 공동자금에 예금한다.

② 만일 그렇지 않다면 카나다잔돈을 정점에 예금한다.

보조정리 8-4.

임의의 find(v)에 대하여 공동자금이나 정점에 예금된 잔돈의 총수는 정확히 v로부터 뿌리까지의 경로상에 있는 매듭들의 개수와 같다.

증명:

명백하다.

결국 해야 할것은 규칙 ①로 예금된 미국잔돈전부를 규칙 ②로 예금된 카나다잔돈전부와 합하는것이다.

기껏해서 M개의 find들을 수행하고 있으므로 find를 수행하는 동안 공동자금에 예금될수 있는 잔돈들의 수를 한정하여야 한다.

보조정리 8-5.

전체 알고리듬에서 규칙 ①에 의한 미국잔돈들의 총 예금은 최대로 $M(G(N)+2)$이다.

증명:

이것은 쉽다. 임의의 find에 대하여 뿌리와 그의 자식때문에 두개의 미국잔돈들이 예금된다. 보조정리 8-3에 의하여 경로우의 정점들은 위수로 단조증가하고 있다. 그리고 최대로 $G(N)$개의 위수그룹들이 있다는데로부터 경로상에 있는 $G(N)$개의 다른 정점들만이 임의의 특정한 find에 대하여 규칙 ①의 예금을 적용할수 있다. 따라서 어떤 한개의 find가 실행될 때 최대로 $G(N)+2$개의 미국잔돈들이 공동자금에 배치될수 있다. 때문에 최대로 $M(G(N)+2)$개의 미국잔돈들이 M개의 find렬에 대하여 규칙 ①로 예금될수 있다.

규칙 ②에 의한 카나다예금전부를 잘 평가하자면 find명령들대신에 정점의 예금들을 합한다. 만일 돈이 규칙 ②로 정점 v에 예금된다면 v는 경로압축에 의해 이동되여 본래의 부모보다 더 높은 위수를 가지는 새로운 부모를 얻을것이다(이것은 경로압축이 진행되고 있다는 사실을 리용하는 경우이다.). 결국 위수그룹 $g>0$에 속하는 정점 v는 그의

부모가 크기때문에 위수그룹 g에서 밀려 나기전에 최대로 F(g)-F(g-1)번 이동될수 있다.[27] 그이후 v에 대한 앞으로의 모든 예금들은 규칙 ①로 진행된다.

보조정리 8-6.

위수그룹 $g>0$에 속하는 정점들의 개수 V(g)는 최대로 $N/2^{F(g-1)}$이다.

증명:

보조정리 8-2에 의하여 위수 r인 정점들은 최대로 $N/2^r$개 있다. 그룹 g에 속하는 위수들을 합하면 다음과 같은것을 얻는다.

$$
\begin{aligned}
V(g) &\leq \sum_{r=F(g-1)+1}^{F(g)} \frac{N}{2^r} \\
&\leq \sum_{r-F(g-1)+1}^{\infty} \frac{N}{2^r} \\
&\leq N \sum_{r=F(g-1)+1}^{\infty} \frac{1}{2^r} \\
&\leq \frac{N}{2^{F(g-1)+1}} \sum_{s=0}^{\infty} \frac{1}{2^s} \\
&\leq \frac{2N}{2^{F(g-1)+1}} \\
&\leq \frac{N}{2^{F(g-1)}}
\end{aligned}
$$

보조정리 8-7.

위수그룹 g의 모든 정점들에 예금된 카나다잔돈들의 최대수는 기껏해서 NF(g)/$2^{F(g-1)}$이다.

증명:

위수그룹안의 매 정점은 그의 부모가 자기 위수그룹에 있는 동안 기껏해서 $F(g)$-$F(g$-1)≦$F(g)$개의 카나다잔돈을 받을수 있으며 보조정리 8-6은 이런 정점들이 몇개나 있는가를 말해 준다. 그 결과는 단순한 곱하기에 의해서 얻어 진다.

---

[27] 이것은 1씩 감소될수 있다. 이 한계는 여기에 아무리 주의를 돌린다 해도 달라 지지 않는다.

보조정리 8-8.

규칙 ②에 의한 총 예금은 카나다잔돈으로 기껏해서 $N\sum_{g=1}^{G(N)} F(g)/2^{F(g-1)}$ 이다.

증명:

위수그룹 0은 위수 0인 요소들만 포함하기때문에 그것은 규칙 ②로 예금할수 없다(그것은 같은 위수그룹안에서 부모를 가질수 없다.). 한계는 다른 위수그룹들을 합하여 얻어 진다.

결국 규칙 ①과 ②에 의하여 예금들을 가진다. 총량은 다음과 같다.

$$M(G(N)+2)+N\sum_{g=1}^{G(N)} F(g)/2^{F(g-1)}$$

$G(N)$ 혹은 그의 거꿀 $F(N)$은 아직 확정하지 않았다. 바라는것을 임의로 선택할수는 있겠지만 우에서의 한계를 최소화하기 위해서는 $G(N)$을 잘 선택하는것이 기본이다. 그러나 $G(N)$이 너무 작으면 $F(N)$은 한계를 넘게 클것이다. 외견상 $F(0)=0$과 $F(i)=2^{F(i-1)}$로 정의되는 재귀함수 $F(i)$를 선택하는것이 좋다. 이것은 $G(N)=1+\lfloor \log^* N \rfloor$을 준다. 표 8-2에서는 위수를 분할하는 방법을 보여 주었다. 그룹 0은 앞의 보조정리에서 요구한것처럼 위수 0만을 포함한다는것을 명심하시오. F는 단일변수악커만함수와 매우 류사하며 다만 토대경우에 대한 정의에서만 차이난다($F(0)=1$).

정리 8-1

M개의 union들과 find들의 실행시간은 $O(M\log^* N)$이다.

증명:

식 8-1에 F와 G의 정의들을 대입한다. 미국잔돈의 총수는 $O(MG(N))=O(M\log^* N)$이며 카나다잔돈의 총수는 $N\sum_{g=1}^{G(N)} F(g)/2^{F(g-1)} = N\sum_{g=1}^{G(N)} 1 = NG(N) = O(N\log^* N)$ 이다. $M=\Omega(N)$이므로 한계는 다음과 같다.

분석은 경로압축에 의하여 자주 이동되는 매듭들이 적으며 따라서 소모되는 총 시간은 비교적 작다는것을 보여 준다.

표 8-2. 증명에서 리용되는 그룹들의 위수분할

그룹	위수
0	0
1	1
2	2
3	3, 4
4	5 ~ 16
5	17 ~ ^{216}
6	65537 ~ 2^{65536}
7	방대한 위수들

제7절. 응용

union/find자료구조의 리용실례는 그림 8-11에서 보여 준것과 같은 미로(한번 들어갔다 빠져 나오기 힘든 길)들의 생성이다. 그림 8-11에서 시작점은 왼쪽 웃구석에 있고 끝점은 오른쪽 아래구석에 있다. 미로는 50*88개의 직4각형세포들을 가지는데 왼쪽 웃구석은 오른쪽 아래구석과 련결되여 있고 세포들은 벽을 통해 이웃세포와 분리되여 있다.

미로를 생성하는 단순한 알고리듬은 어디에서나 벽으로부터 시작한다(입력과 출력에서는 례외이다.). 다음 계속하여 벽을 임의로 선택하고 그 벽이 분리하는 세포가 아직 다른것과 련결되여 있지 않으면 그 벽을 헐어 버린다. 시작과 끝세포들이 련결될 때까지 이 처리를 반복하면 미로를 얻는다. 실제로 매 세포가 다른 세포로부터 도달할수 있을 때까지 벽들을 계속 헐어 버리는것이 더 좋다(이것은 미로에서 더 많은 가짜통로(false lead)들을 생성한다.).

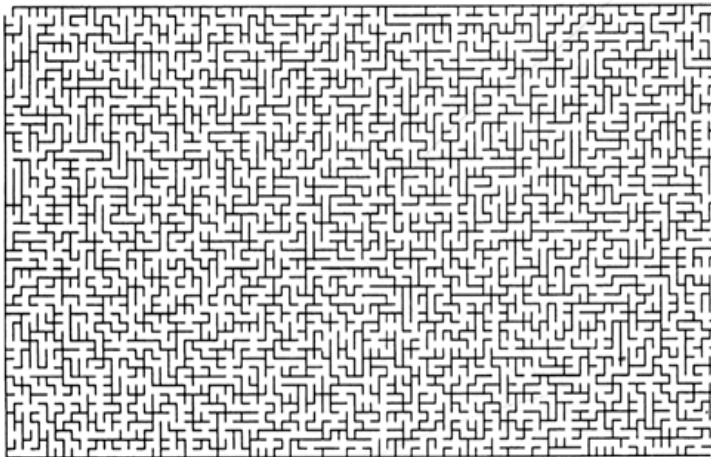



그림 8-11. 50*88미로

5*5미로에 대한 알고리듬을 례들어 설명한다. 그림 8-12에서는 초기구성을 보여 주었다. 서로 련결되여 있는 세포들의 모임을 표현하는데 union/find자료구조를 리용한다. 초기에 벽들은 도처에 있으며 매 세포는 그자체의 등가클라스내에 있다.

{1} {2} {3} {4} {5} {6} {7} {8} {9} {10} {11} {12} {13} {14} {15} {16} {17} {18} {19} {20} {21} {22} {23} {24}

그림 8-12. 초기상태: 모든 벽들이 세워 지고 모든 세포들은 그자체의 모임에 있다.

{0, 1} {2} {3} {4, 6, 7, 8, 9, 13, 14} {5} {10, 11, 15} {12} {16, 17, 18, 22} {19} {20} {21} {23} {24}

그림 8-13. 알고리듬의 어떤 단계에서: 몇개의 벽들이 허물어 지고 모임들이 병합되였다. 이 단계에서 8과 13사이의 벽이 우연히 선택된다면 8과 13은 이미 련결되였으므로 그 벽을 헐어 버리지 않는다.

{0, 1} {2} {3} {4, 6, 7, 8, 9, 13, 14, 16, 17, 18, 22} {5} {10, 11, 15} {12} {19} {20} {21} {23} {24}

그림 8-14. 그림 8-13에서 바른4각형 18과 13사이의 벽이 선택되면 18과 13은 아직 련결되지 않았으므로 벽이 제거되고 그 모임들은 병합된다.

{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24}

그림 8-15. 결국 24개의 벽들이 제거되고 모든 요소들은 같은 모임에 속한다.

알고리듬의 실행시간은 union/find비용에 의해 지배된다. union/find계의 크기는 세포들의 개수와 같다. 제거되는 벽의 개수는 세포의 개수보다 하나 작으므로 find연산의 수는 세포들의 수에 비례한다. 그러나 주의하여 보면 벽의 개수가 처음 위치에 있는 세포들보다 약 2배일뿐이라는것을 알게 된다. 따라서 만일 세포개수가 N이라면 우연히 선택

된 벽마다 2개의 find를 가지기때문에 알고리듬전과정에 $2N \sim 4N$개의 find연산을 가진다고 추정할수 있다. 그러므로 알고리듬의 실행시간은 $O(N\log^*N)$이며 이 알고리듬은 미로를 빨리 생성한다.

요약

분리모임들을 보존하는 매우 단순한 자료구조를 보았다. union연산이 수행될 때 모임이 이름을 정확히 보유하고 있는가 하는것은 문제될것이 없다. 여기서 명심해야 할것은 특별한 단계가 전혀 지정되지 않았을 때 심사숙고하여 방법을 택하는것이 매우 중요하다는것이다. union단계는 이러한 우점을 리용하여 유연해 질수 있으며 훨씬 더 능률적인 알고리듬을 얻을수 있다.

경로압축은 다른데서(펼친나무(splay tree), 비대칭더미(skew heap) 등) 볼수 있는 자동조정(self-adjustment)의 가장 쉬운 형식들중의 하나이다. 그 리용은 특히 리론적면에서 매우 흥미 있다. 그것은 그리 단순치 않은 최악의 경우에 대한 분석을 가진 단순한 알고리듬의 최초의 실례들중의 하나이기때문이다.

련습문제

8-1. 다음과 같은 명령렬의 결과를 보여 주시오.
union(1,2), union(3,4) union(3,5) union(1,7)
union(3,6), union(8,9) union(1,8) union(3,10)
union(3,11), union(3,12) union(3,13) union(14,15)
union(16,0), union(14,16) union(1,3) union(1,14)
이때 union들은:
ㄱ. 임의로 수행된다.
ㄴ. 높이에 의해 수행된다.
ㄷ. 크기에 의해 수행된다.

8-2. 련습문제 8-1에서 매 나무들에 대하여 가장 깊은 매듭에 대한 경로압축을 가지고 find를 수행하시오.

8-3. 경로압축의 효과들과 여러가지 union전략들을 결정하는 프로그람을 쓰시오. 이 프로그람은 모두 6개의 가능한 전략들을 리용해서 등가연산들의 긴렬을 처리해야 한다.

8-4. union들이 높이에 의해 수행되면 임의의 나무의 깊이는 $O(\log N)$이라는것을 보여 주시오.

8-5. ㄱ. $M=N^2$이면 M개의 union/find연산들의 실행시간은 $O(M)$이라는것을 보여 주

시오.

ㄴ. $M=N\log N$이면 M개의 union/find연산들의 실행시간은 $O(M)$이라는것을 보여주시오.

ㄷ. $M=\Theta(N\log\log N)$라고 하자. M개의 union/find연산들의 실행시간은 얼마인가?

ㄹ. $M=\Theta(N\log^* N)$라고 하자. M개의 union/find연산들의 실행시간은 얼마인가?

8-6. 제8장 제7절의 알고리듬에 의해 생성된 미로들에 대하여 시작부터 끝점까지의 경로는 유일하다는것을 증명하시오.

8-7. 시작부터 끝까지 어떤 경로도 포함하지 않으나 미리 지정된 벽의 제거는 유일한 경로를 생성하는 속성을 가지는 미로를 발생하는 알고리듬을 설계하시오.

8-8. 취소되지 않은 최후의 union연산을 취소하는 여분의 연산 deunion을 추가하려 한다고 하자.

ㄱ. 높이에 의한 union과 find를 경로압축이 없이 수행하면 deunion은 쉬우며 M개의 union, find, deunion연산렬은 $O(M\log N)$시간을 가진다는것을 보여주시오.

ㄴ. 왜 경로압축이 deunion을 어렵게 하는가?

ㄷ. M개의 연산렬이 $O(M\log N/\log\log N)$시간을 가지도록 세개의 연산을 실행하는 방법을 보여 주시오.

8-9. 현재 x가 속하는 모임에서 x를 제거하여 자체의 모임에 넣는 여분연산 remove(x)를 추가하려 한다고 하자. M개의 union, find, remove연산렬의 실행시간이 $O(M\alpha(M,N))$이 되도록 union/find알고리듬을 변경하는 방법을 보여주시오.

8-10. 입력으로서 N-정점나무와 N개 정점쌍들의 목록을 가지고 매 쌍(v, w)에 대하여 v와 w의 가장 가까운 공통선조를 결정하는 알고리듬을 작성하시오. 이 알고리듬은 $O(N\log^* N)$으로 실행해야 한다.

8-11. union들 모두가 find보다 앞선다면 경로압축을 가지는 **분리모임알고리듬**은 union들이 임의로 수행되여도 선형시간을 요구한다는것을 보여 주시오.

8-12. union들이 임의로 수행된다면(그런데 경로압축은 find들에 대하여 수행된다.) 최악의 경우 실행시간은 $\Theta(M\log N)$이라는것을 증명하시오.

8-13. union들이 크기에 의해 수행되고 경로압축이 실행된다면 최악의 경우 실행시간은 $O(M\log^* N)$이라는것을 증명하시오.

8-14. i로부터 뿌리까지의 경로상에 있는 서로 다른 매듭들을 그의 조부모와 련결시켜 find(i)에 대한 부분적경로압축을 실현한다고 하자. 이것을 **경로등분**(*path halving*)이라고 한다.

ㄱ. 이를 수행하는 수속을 작성하시오.

ㄴ. 경로의 2등분이 find에 대하여 수행되며 크기에 의한 union이나 높이에 의

한 union이 리용된다면 최악의 경우 실행시간은 O(Mlog*N)이라는것을 증명하시오.

8-15. 임의의 크기의 미로들을 생성하는 프로그람을 작성하시오. Visual C++와 같이 창문조작을 가진 체계를 리용하고 있다면 그림 8-11과 류사한 미로를 생성하시오. 만일 그렇지 않다면 미로의 문자적표현(실례로 출력하는 매 행은 정방형을 표현하며 어느 벽들이 존재하는가에 대한 정보를 가진다.)을 서술하고 프로그람으로 그 표현을 생성하게 하시오.

참고문헌

union/find문제에 대한 여러가지 풀이들을 [6], [9], [11]에서 찾아 볼수 있다. Hopcroft와 Ullman은 제8장 제6절의 O(Mlog*N)한계를 보여 주었다. Tarjan[15]는 한계 O($M\alpha(M,N)$)를 얻었다. $M<N$에 대한 더 정확한(그러나 비대칭적으로 항등인) 한계는 [2]와 [18]에서 나온다. 경로압축과 union들에 대한 여러가지 다른 전략들도 같은 한계를 가진다. 상세한것은 [18]을 보시오.

일정한 제한밑에서 M개의 union/find연산들을 처리하는데 $\Omega(M\alpha(M,N))$시간이 요구된다는것을 보여 주는 보다 낮은 한계는 Tarjan[16]에 의해 주어 졌다. 보다 적은 제한조건하에서 동일한 한계들은 [7]과 [14]에서 보여 주었다.

union/find자료구조에 대한 응용은 [1]과 [10]에서 나온다. union/find문제의 어떤 특수경우들은 $O(M)$시간에 풀려 질수 있다[8]. 이것은 [1]과 같이 그라프의 우세와 인수 $\alpha(M,N)$에 의한 가약성(제9장에 있는 참고문헌을 보시오.) 등 여러가지 알고리듬들의 실행시간을 감소시킨다. 이 장에서의 그라프련결성문제와 같은 기타 문제들에는 영향을 주지 않는다[10]. 이 론문은 10개의 실례를 목록화하였다. Tarjan은 여러가지 그라프문제들에 대한 능률적인 알고리듬들을 얻는데 경로압축을 리용하였다.[17]

union/find문제에 대한 평균경우결과들은 [5], [12], [21], [3]에서 나온다. 임의의 단일연산의 실행시간(전체 렬과 대립될 때)을 한계 짓는 결과들은 [4],[13]에서 나온다.

련습문제 8-8은 [20]에 풀려 있다. 더 많은 연산들을 지원하는 일반적인 union/find구조는 [19]에 주어 진다.

1. A. V. Aho, J. E. Hopcroft, and J. D. Ullman, "On Finding Lowest Common Ancestors in Trees," *SIAM journal on Computing,* 5 (1976), 115-132.
2. L. Banachowski, "A Complement to Tarjan's Result about the Lower Bound on the Complexity of the Set Union Problem," *Information Processing Letters,* 11 (1980), 59-65.
3. B. Bollobas and I. Simon, "Probabilistic Analysis of Disjoint Set Union Algorithms," *SIAM Journal on Computing,* 22 (1993), 1053-1086.
4. N. Blum, "On the Single-Operation Worst-Case Time Complexity of the Disjoint Set Union

Problem," *SIAM Journal on Computing,* 15 (1986), 1021-1024.

5. J. Doyle and R. L. Rivest, "Linear Expected Time of a Simple Union Find Algorithm," *Information Processing Letters, 5* (1976), 146-148.
6. M. J. Fischer, "Efficiency of Equivalence Algorithms," in *Complexity of Computer Computation* (eds. R. E. Miller and J. W. Thatcher), Plenum Press, New York 1972, 153-168.
7. M. L. Fredman and M. E. Saks, "The Cell Probe Complexity of Dynamic Data Structures," *Proceedings of the Twenty-first Annual Symposium on Theory of Computing* (1989), 345-354.
8. H. N. Gabow and R. E. Tarjan, "A Linear-Time Algorithm for a Special Case of Disjoint Set Union," *Journal of Computer and System Sciences,* 30 (1985), 209-221.
9. B. A. Galler and M. J. Eischer, "An Improved Equivalence Algorithm," *Communications of the ACM,* 7 (1964), 301-303.
10. J. E. Hopcroft and R. M. Karp, "An Algorithm for Testing the Equivalence of Finite Automata," *Technical Report TR-*71-114*,* Department of Computer Science, Cornell University, Ithaca, N.Y., 1971.
11. J. E. Hopcroft and J. D. Ullman, "Set Merging Algorithms," *SIAM Journal on Computing,* 2 (1973), 294-303.
12. D. E. Knuth and A. Schonhage, "The Expected Linearity of a Simple Equivalence Algorithm," *Theoretical Computer Science,* 6 (1978), 281-315.
13. J. A. LaPoutre, "New Techniques for the Union-Find Problem," *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms* (1990), 54-63.
14. J. A. LaPoutre, "Lower Bounds for the Union-Find and the Split-Find Problem on Pointer Machines," *Proceedings of the Twenty-Second Annual ACM Symposium on Theory of Computing* (1990), 34-44.
15. R. E. Tarjan, "Efficiency of a Good but Not Linear Set Union Algorithm," *Journal of the ACM,* 22 (197*5),* 215-225.
16. R. E. Tarjan, "A Class of Algorithms Which Require Nonlinear Time to Maintain Disjoint Sets," *Journal of Computer and System Sciences,* 18 (1979), 110-127.
17. R. E. Tarjan, "Applications of Path Compression on Balanced Trees," *Journal of the ACM,* 26 (1979), 690-715.
18. R. E. Tarjan and J. van Leeuwen, "Worst Case Analysis of Set Union Algorithms,*" Journal of the ACM,* 31 (1984), 245-281.
19. M. J. van Kreveld and M. H. Overmars, "Union-Copy Structures and Dynamic Segment Trees," *Journal of the ACM,* 40 (1993), 635-652.
20. J. Westbrook and R. E. Tarjan, "Amortized Analysis of Algorithms for Set Union with Backtracking," *SIAM Journal on Computing,* 18 (1989), 1-11.
21. A. C. Yao, "On the Average Behavior of Set Merging Algorithms," *Proceedings of Eighth Annual ACM Symposium on the Theory of Computation,* (1976), 192-195.

제9장. 그라프알고리듬

이 장에서는 그라프리론에서 나서는 여러가지 공통적인 문제들을 설명한다. 이 알고리듬들은 실천적으로 쓸모 있을뿐아니라 자료구조의 선택에 세심한 주의를 돌리지 않으면 많은 실천적응용에서 속도가 크게 떠지므로 흥미가 있다.

이 장에서는 다음과 같은 내용을 고찰한다.

- 그라프문제로 바꿀수 있는 여러가지 실천적문제들의 제시
- 여러가지 일반적인 그라프문제들을 해결하기 위한 알고리듬
- 알고리듬의 실행시간을 크게 줄일수 있는 적합한 자료구조의 선택방법
- 깊이우선탐색이라고 하는 중요한 수법에 대하여 고찰하고 그것이 표면상 중요한 여러가지 문제들을 선형시간내에 해결하는데 어떻게 리용되는가를 본다.

제1절. 정의

그라프(*graph*) $G=(V, E)$는 **정점**(*vertices*)들의 모임 V와 **변**(*edges*)들의 모임인 E로 이루어 져 있다. 매 변은 (v, w)의 쌍이며 여기서 $v, w \in V$이다. 때때로 변을 **호**(*arc*)라고도 한다. 그 쌍이 순서 있는 쌍이면 그 그라프는 방향을 가진다. 방향을 가진 그라프를 때때로 **방향그라프**(*digraphs*)라고도 한다. $(v, w) \in E$이면 정점 w는 v와 린접되여 있다. 변(v, w)를 가진 무방향그라프에서는 w가 v와 린접되여 있고 v는 w와 린접되여 있다. 때때로 변은 세번째 요소를 가지는데 이것을 **무게**(*weight*) 혹은 **값**(*cost*)이라고 한다.

그라프에서 **경로**(*path*)는 $1 \leq i < N$ 에 대하여 $(w_i, w_{i+1}) \in E$인 정점 $w_1, w_2, \ldots, w_N$의 렬이다. 여기에서 경로의 **길이**(*length*)는 그 경로변들의 수이며 이것은 $N-1$과 같다. 어떤 정점으로부터 자기자체로 들어 오는 경로에 대해서도 정의하며 만일 이 경로가 변을 가지지 않는다면 경로의 길이는 0이다. 이것은 다른 특수한 경우를 정의하는데 편리하다. 그라프가 어떤 정점에서 나와 그자체에로 들어 오는 변 (v, v)를 포함한다면 때때로 경로v, v를 **순환고리**(*loop*)라고 한다. 여기서 고찰하는 그라프들은 일반적으로 고리가 없는 그라프들이다. **단순경로**(*simple path*)는 첫 정점과 마지막정점이 같을수 있다는것을 제외하고는 모든 정점들이 독립적인 그러한 경로이다.

방향그라프에서 **순환**(*cycle*)은 $w_1=w_N$인 적어도 길이가 1인 경로이다. 즉 경로가 단순하면 이 순환도 단순하다. 무방향그라프들에 관해서는 변들을 명백히 표현할것을 요구한다. 이에 관한 론리적요구는 무방향그라프에서 경로 u, v, u가 하나의 순환으로 고찰되지 말아야 할것이다. 왜냐하면 그것은 (u, v)와 (v, u)가 같은 변이기때문이다. 방향그라

프에서 이것들은 서로 다른 변들이므로 이것을 순환이라고 한다. 방향그라프가 순환을 가지지 않는다면 **비순환**(*acycle*)적이다. **방향성비순환그라프**를 때때로 줄임말 DAG로 나타낸다.

매 정점으로부터 그밖의 모든 정점에로 경로를 가진다면 무방향그라프는 련결그라프로 된다. 이러한 속성을 가진 방향그라프를 **강한 련결그라프**(*strongly connected*)라고 한다. 만일 방향그라프가 강한 련결은 아니지만 호들에 대한 방향이 없이 그라프가 모호하며 련결되였다면 그 그라프는 **약한 련결그라프**(*weakly connected*)라고 한다. **완전그라프**(*complete graph*)는 매 쌍의 두 정점들사이에 변이 모두 존재하는 그라프이다.

실생활에서 그라프로 모형화할수 있는 전형적인 실례는 항공체계이다. 매 비행장은 하나의 정점이고 정점으로 표현된 비행장들로부터 무착륙비행항로가 있다면 두 정점들을 변으로 련결한다. 변은 비행시간, 비행거리 혹은 비행비용을 나타내는 무게를 가질수 있다. 그 무게는 각이한 방향으로 비행하는데 따라 비용이 더 들거나 시간이 더 걸릴수 있으므로 이러한 그라프는 방향성으로 보는것이 옳다. 임의의 비행장에서 다른 비행장에로의 비행이 항상 가능하므로 항공체계는 강한 련결그라프로 모의할수 있다. 또한 임의의 두 비행장사이의 최단비행을 빨리 결정할수도 있다. 여기서 《최단》이라는것은 최소한의 변의 수를 가진 경로를 의미하든지 아니면 무게가 제일 작은 경로에 대하여 쓰는 말이다.

교통체계도 그라프로 모형화할수 있다. 매 거리교차점은 정점을 나타내며 매 거리는 변을 나타낸다. 변의 무게는 여러가지가운데서 속도제한 혹은 통과능력(주로선의 수)을 나타낼수 있다. 그다음 최단경로를 요구하거나 가장 좋은 통로의 위치를 찾는데 이 정보를 리용할수 있다.

이 장의 마지막부분에서 그라프에 관한 여러가지 응용들을 본다. 이 그라프들중 대다수는 그 규모가 매우 크다. 따라서 사용하는 알고리듬들이 매우 효률적이여야 한다.

1. 그라프의 표현

방향그라프를 고찰하여 보자(무방향그라프는 간단히 표현된다.).

지금 1에서 시작하여 정점들의 수를 셀수 있다고 하자. 그림 9-1에서 보여 준 그라프는 7개의 정점과 12개의 변을 나타낸다.

그라프를 표현하는 가장 간단한 방법은 2차원배렬을 리용하는것이다. 이것을 **린접행렬**(*adjacency matrix*)표시법이라고 한다. 매 변 (u, v)에 대하여 $A[u][v]$를 true로 설정하며 그렇지 않다면 배렬내에 있는 요소들은 false이다. 변이 그와 관련된 무게를 가진다면 $A[u][v]$를 무게값과 같게 설정하며 존재하지 않는 변들을 나타내기 위한 표식으로서 매우 큰 무게값 혹은 매우 작은 무게값을 리용할수 있다. 실례로 값 눅은 비행항로를 찾아 본

다면 존재하지 않는 비행항로를 ∞로 표현할수 있다. 값 비싼 비행항로를 찾아 본다면 존재하지 않는 변을 나타내는데 -∞ 혹은 경우에 따라 0을 리용할수 있다.

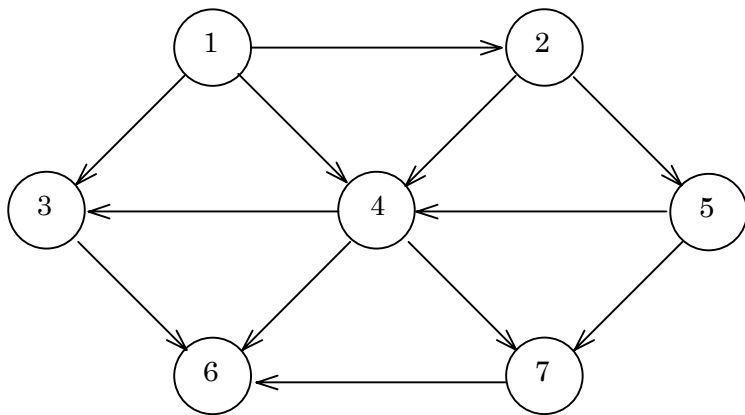



그림 9-1. 방향그라프

이것은 매우 단순하다는 우점을 가지지만 공간요구는 $\Theta(|V|^2)$이므로 그라프가 매우 많은 변을 가지지 않는다면 이러한 표시방법은 아주 비효률적이다. 그라프가 조밀한 그라프라면 즉 $|E|=\Theta(|V|^2)$이라면 린접행렬로 표현하는것이 좋다. 그러나 대부분의 실천적인 응용들에서 이것은 사실과 맞지 않는다. 실례로 그라프가 어떤 거리의 지도를 표시한다고 하자. 맨하탄에서의 방향문제와 같은것을 가정하면 여기서 거의 모든 거리들은 북-남 혹은 동-서로 놓여 있다. 그러므로 대략적으로 4개의 도로들이 교차되는 곳에 임의의 교차점이 생기므로 그라프가 방향을 가지고 있고 모든 거리들이 2개의 길을 가진다면 $|E|\approx4|V|$이다. 3000개의 교차점이 있다면 12000개의 변을 가진 3000정점그라프가 된다. 그리고 이것은 9,000,000크기의 배렬을 요구한다. 이 대부분의 배렬요소들은 0으로 될것이다. 이것은 직관적으로 좋다고 볼수 없다. 왜냐하면 그라프자료구조는 존재하지 않는 자료가 아니라 실제로 존재하는 자료를 표현하기 위한것이기때문이다.

그라프가 조밀하지 않다면 다시말하여 그라프가 성글다면 이에 대한 보다 좋은 해결방법은 **린접목록**(*adjacency list*)표현을 리용하는것이다. 매 정점에 대하여 모든 린접정점목록을 만드는데 그때 공간요구조건은 $O(|E|+|V|)$이며 그라프의 크기는 선형이다. 그림 9-2에서 제일 왼쪽 구조는 순수 선두매듭들의 배렬이다. 이 표현방법은 그림 9-2로부터 명백하다. 변이 무게를 가진다면 이와 관련한 보충적인 정보가 매듭에 보관된다.

린접목록은 그라프를 나타내는 표준방식이다. 무방향그라프도 이와 류사하게 표현할 수 있다. 매 변 (u, v)는 2개의 목록에 나타나므로 무방향그라프에 대한 기억기리용은 본질적으로 방향그라프의 두배로 된다. 그라프알고리듬에서 일반적인 요구는 어떤 주어 진

정점 v와 린접하는 모든 정점들을 찾아 내는것이며 이러한 처리는 알맞는 린접목록을 정점들의 수에 시간비례하여 간단히 훑어 봄으로써 해결할수 있다.

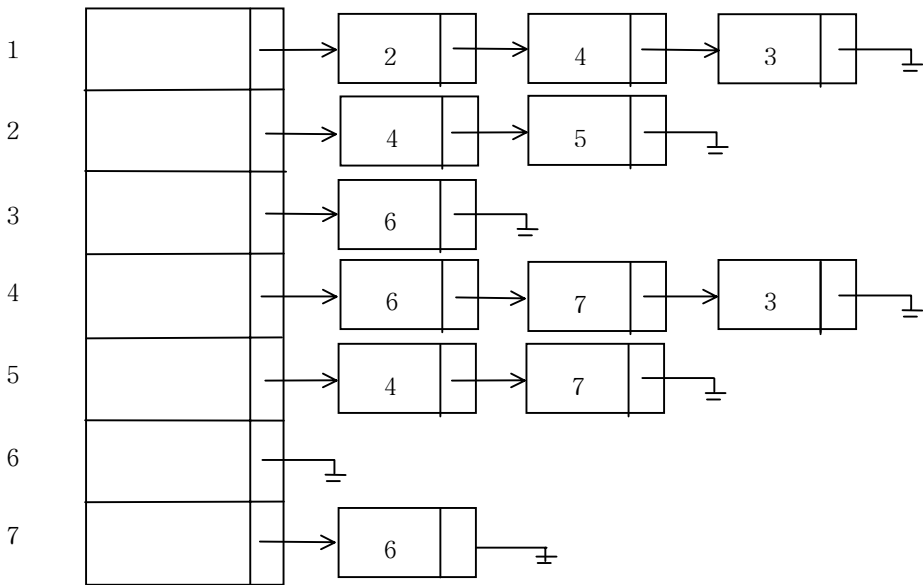



그림 9-2. 그라프의 린접목록표현

린접정점들의 목록을 포함하여 매개 정점에 대한 정보는 vertex형의 대상에 보관된다. 대다수 실천적응용에서 정점들은 이름을 가지며 번역시에는 알려 져 있지 않다. 따라서 일반적으로 그의 대응하는 vertex객체에 이름을 대응시킬 필요가 있다. 이것을 실현하는 가장 간단한 방법은 하쉬표를 리용하는것이다. 하쉬표에 열쇠로 되는 이름과 vertex를 가리키는 지적자를 보관한다. 그라프를 읽어 들일 때 새로운 vertex객체들이 창조된다. 매 변의 입력과 동시에 그 두개 정점이 이미 있는가를 검사한다. 만약 있다면 그에 대응하는 vertex를 리용한다. 만일 없다면 새로운 vertex객체를 창조하고 하나의 쌍으로서 그 이름과 vertex를 하쉬표에 삽입한다. 매 vertex객체입력점은 또한 정점이름을 보관하고 종국적으로는 이 이름을 출력하는것이 필요할것이다.

이 장에서 서술한 코드는 가능한껏 추상자료형(ADT)을 리용한 가상코드이다. 이것은 알고리듬적인 표현을 보다 더 명백하게 하고 공간을 절약하게 할것이다. 부록 A에는 제9장 제3절 1에서 설명하는 최단경로알고리듬에 관한 작업코드를 주었다. 2개의 판본이 주어 졌는데 하나는 표준형판서고(STL)를 리용한것이고 다른 하나는 앞의 장들에서 개발된 자료구조들을 리용한것이다.

제2절. 위상학적정렬

위상학적정렬(*topological sort*)은 방향성비순환그라프에서 정점들의 순서붙이기이다. 즉 v_i로부터 v_j에로 경로가 없다면 v_i뒤에 v_j가 오도록 순서를 붙인다. 그림 9-3에서 보여준 그라프는 마이아미에 있는 국립대학의 필수적인 교육과정구조를 나타낸다. 방향 있는 변 (v, w)는 과정 w가 시작되기전에 완료되여야 한다는것을 의미한다. 이 교육과정에 대한 위상학적순서붙이기는 선행요구를 위반하지 않는 임의의 교육과정순서이다.

그라프가 하나의 순환을 가진다면 위상학적순서붙이기는 불가능하다는것이 명백하다. 그것은 순환우에서 두개의 정점 v와 w에 대하여 v는 w에 앞서며 w는 v에 앞서기때문이다. 게다가 순서붙이기는 필연적으로 유일하지 않다. 그림 9-4의 그라프에서 $v_1, v_2, v_5, v_4, v_3, v_7, v_6$과 $v_1, v_2, v_5, v_4, v_7, v_3, v_6$은 둘다 위상학적인 순서붙이기이다.

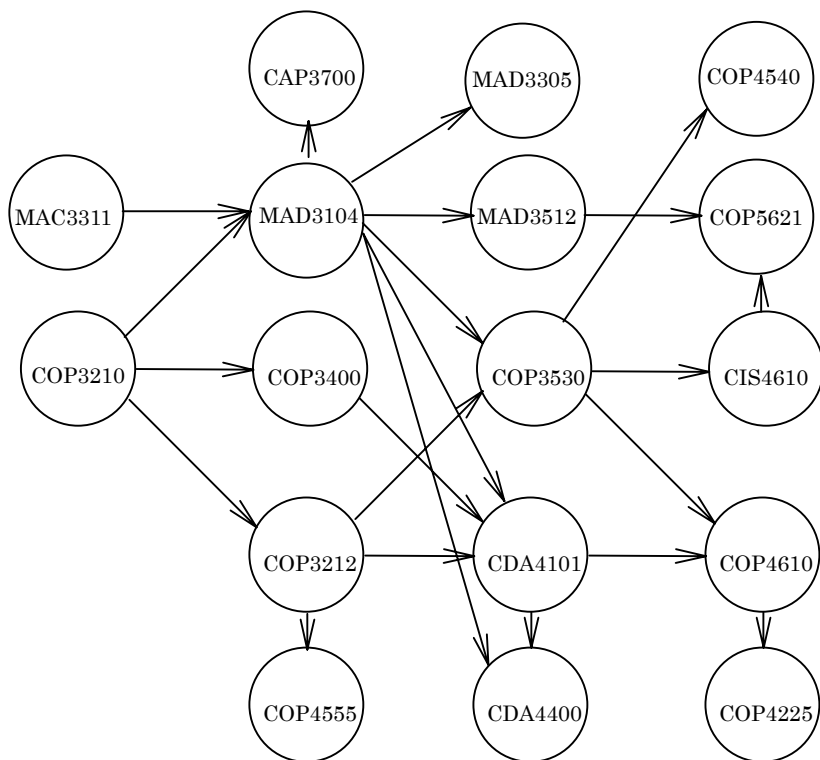



그림 9-3. 필수교육과정구조를 나타내는 비순환그라프

위상학적순서를 찾는 간단한 알고리듬은 먼저 들어 오는 변이 하나도 없는 임의의 정점을 찾는것이다. 그다음 그라프에서 그 변들에 따라서 이 정점을 출력하고 그것을 제

거한다. 그다음 나머지그라프에 대하여 이와 같은 방법을 적용한다.

이것을 형식화하기 위하여 변 (u, v)들의 개수로서 정점 v의 입력도수(*indegree*)를 정의한다. 그라프에서 모든 정점들의 입력도수를 계산한다. 매 정점들의 입력도수가 보관되여 있고 그 그라프를 린접목록에로 읽어 들인다고 하면 그다음 위상구조순서붙이기를 진행하기 위하여 그림 9-5의 알고리듬을 적용할수 있다.

함수 findNewVertexOfIndegreeZero는 정점들의 배렬을 주사하여 이미 위상학적번호를 할당받지 못한 입력도수가 0인 정점을 찾는다. 이런 정점이 존재하지 않는다면 그것은 NOT_A_VERTEX에 돌아 가는데 이것은 그라프가 하나의 순환을 가진다는것을 의미한다.

findNewVertexOfIndgreeZero는 정점들의 배렬을 순차적으로 간단히 주사하므로 그에 대한 매개 호출은 $O(|V|)$시간이 걸린다. 이러한 호출이 $|V|$개 있으므로 알고리듬의 실행시간은 $O(|V|^2)$으로 된다.

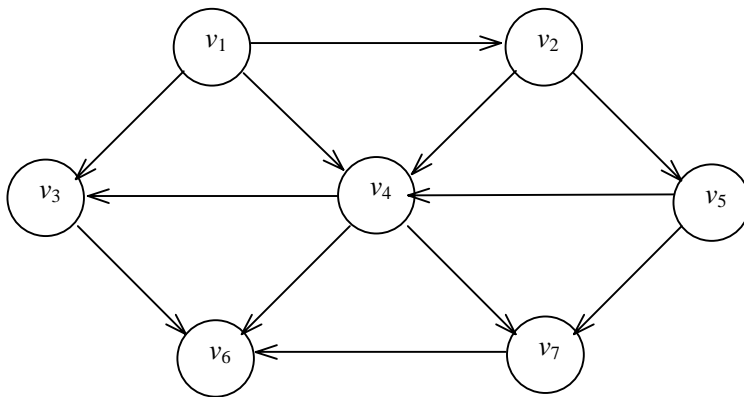



그림 9-4. 비순환 그라프

```
void Graph::topsort( )
{
    Vertex v, w;
    for( int counter = 0; counter < NUM_VERTICES; counter++ )
    {
        v = f'indNewVertexOfDegreeZero( );
        if( v == NOT_A_VERTEX )
            throw Cyc1eFound
        v.topNum = counter;
        for each w adjacent to v
            w.indegree--;
    }
}
```

프로그람 9-1. 간단한 위상학적정렬가상코드

자료구조에 대하여 더 주의깊게 고찰해 보면 위상학적정렬을 보다 더 좋게 실현할 수 있다. 우의 산법에서 실행시간이 불충분한 원인은 정점들의 배렬을 순서대로 주사하는데 있다. 그라프가 성글다면 매 순환과정에 극소수의 정점만 갱신되는 입력도수를 가진다는것을 알수 있다. 그렇지만 입력도수가 0인 어떤 정점을 찾아 내는데 비록 극소수의 변화가 있다고 해도 모든 정점들을(포텐샬적으로) 찾는다.

따라서 입력도수가 0인 모든 정점들을 특수한 통(box)에 보관함으로써 이러한 비효률성을 없앨수 있다. findNewVertexOfIndegreeZero함수는 그때 통에서 임의의 정점을 돌려준다. 린접점의 입력도수가 감소하면 매개 정점을 검사하여 그 입력도수가 0으로 떨어질 때 통안에 그것을 넣는다.

통을 실현하기 위하여 탄창이나 대기렬을 리용할수 있는데 여기서는 대기렬을 리용한다. 먼저 모든 정점에 대하여 입력도수가 검사된다. 그다음 입력도수가 0인 모든 정점들이 초기의 빈 대기렬에 넣어 진다. 대기렬이 비지 않는 동안 정점 v는 제거되며 v와 린접한 모든 변들이 감소된 입력도수를 가진다. 그 입력도수가 0으로 떨어 지자마자 정점은 대기렬에 놓인다. 그때 위상학적순서는 쌍방향대기렬에 놓여 있는 정점들의 순서이다. 표 9-1은 매 단계에서의 상태를 보여 준다.

이 알고리듬에 관한 가상코드를 프로그람 9-2에 주었다. 앞에서와 같이 그라프가 이미 린접표에 넣어 져 있고 입력도수는 정점들에 의하여 계산되여 보관된다고 하자. 또한 매 정점이 그것의 위상학적번호가 있는 topNum이라고 하는 자료성원을 가진다고 하자.

린접표를 리용하면 이 알고리듬을 실행하는데 걸리는 시간은 $O(|E|+|V|)$이다. 이것은 for순환체가 매변에 대하여 기껏해서 한번 실행된다는것을 알면 명백하다. 대기렬연산은 매 정점에 대하여 기껏해서 한번은 수행되며 초기화단계는 그라프의 크기에 비례하는 시간이 걸린다.

표 9-1. 그림 9-4에서 보여 준 그라프에 대한 위상학적정렬을 적용한 결과

	쌍방향대기렬에 들어 가기전의 입구도수 #						
정점	**1**	**2**	**3**	**4**	**5**	**6**	**7**
v	0	0	0	0	0	0	0
v	1	0	0	0	0	0	0
v	2	1	1	1	0	0	0
v	3	2	1	0	0	0	0
v	1	1	0	0	0	0	0
v	3	3	3	3	2	1	0
v	2	2	2	2	0	0	0
대기렬들어 가기	v_1	v_2	v_5	v_4	v_3, v_7		v_6
대기렬에서 나오기	v_1	v_2	v_5	v_4	v_3	v_7	v_6

```
void Graph::topsort( )
{
   Queue q( NUM_VERTICES );
   int counter = 0;
   Vertex v, w;

/*1*/      q.makeEmpty( );
/*2*/      for each vertex v
/*3*/      if( v.indegree = = 0 )
/*4*/            q. enqueue ( v );
/*5*/      while( !q.1sEmpty( ) )
           {
/*6*/            v = q.dequeue( );
/*7*/            v.topNum = ++counter; // Assign next number

/*8*/            for each w adjacent to v
/*9*/               if( --w.indegree = = 0 )
/*10*/                    q.enqueue( w );
}

/*1l*/     if( counter != NUM_VERTICES )
/*12*/         throw Cyc1eFound( );
}
```

프로그람 9-2. 위상학적정렬을 진행하기 위한 가상코드

제3절. 최단경로알고리듬

이 절에서는 여러가지 최단경로문제를 고찰한다. 입력자료는 무게붙은그라프인데 매 변(v_i, v_i)과 관련되는것은 그 변을 지나는데 드는 값 (i, j)이다. 경로 v_1, v_2,…, v_N의 값은 $\sum_{i=1}^{N-1} c_{i,i+1}$ 이다. 이것을 **무게붙은경로길이**(*weighted path length*)라고 한다. **무게붙지않은 경로길이**(*unweighted path length*)는 순수 경로상의 변들의 개수이다. 즉 N−1이다.

단일원천최단경로문제

무게붙은그라프 $G = (V, E)$와 구별되는 정점 s가 입력으로 주어 졌을 때 G에서 s로부터 다른 모든 정점에 이르는 무게붙은최단경로를 찾으시오.

실례로 그림 9-5에서 보여 준 그라프에서 v_1부터 v_6에 이르는 무게붙은최단경로는 값 6을 가지며 v_1로부터 $v_4 \rightarrow v_7 \rightarrow v_6$에 이른다. 이 정점들사이에서 무게를 가지지 않는 최단경로는 2이다. 일반적으로 무게를 가진 경로인가 무게를 가지지 않는 경로인가를 특별

히 지적하지 않으면 그 그라프의 경로는 무게를 가진것으로 고찰한다. 또한 이 그라프에는 v_6으로부터 v_1에 이르는 경로가 없다는것을 알수 있다.

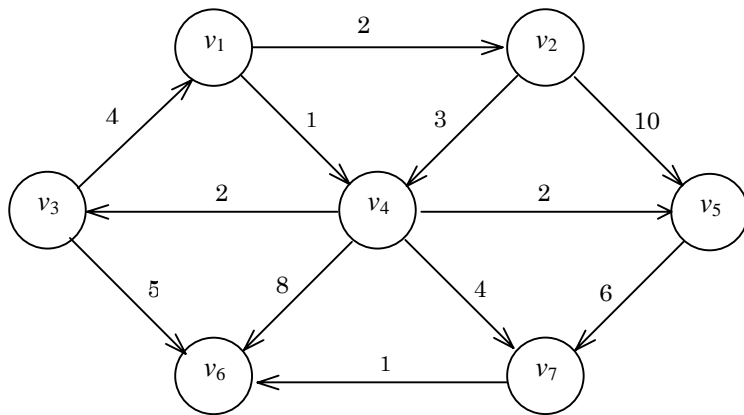



그림 9-5. 방향그라프 *G*

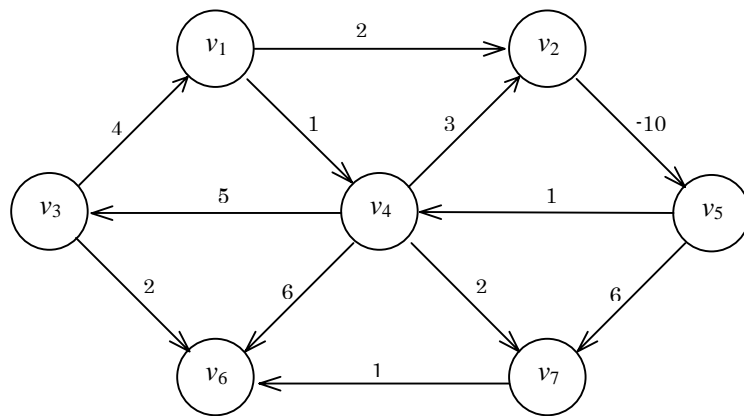



그림 9-6. 부값순환그라프

앞의 실례에서 그라프는 부값변들을 가지지 않는다. 그림 9-6의 그라프는 부값변이 존재하는 경우의 문제를 보여 준다. v_5로부터 v_4에 이르는 경로는 값 1을 가지지만 보다 더 짧은 경로는 값이 -5인 순환고리 v_5, v_4, v_2, v_5, v_4이다. 이 경로는 여전히 최단경로가 아니다. 그것은 멋없이 긴 순환고리이기때문이다. 따라서 이 두 점사이의 최단경로는 정의되지 않는다. 이와 마찬가지로 v_1로부터 v_6에 이르는 최단경로는 정의되지 않는데 그것은 같은 순환고리이기때문이다. 이 순환고리를 **부값순환**(*negative-cost cycle*)이라고 하는데 그라프내에 이런것이 존재하면 최단경로는 정의되지 않는다. 부값변들은 순환이 있으면 반드시 좋지 않은 문제를 일으키며 문제를 더 어렵게 한다. 편리상 부값순환이 없는 경

우 s로부터 s에로 이르는 최단경로는 0으로 한다.

최단경로문제를 풀기 위한 많은 실례가 있다. 만일 정점들이 콤퓨터를 나타내고 변들이 콤퓨터들사이의 련결을 나타내며 값이 통신비용(자료의 1000byte당 전화료금),지연비용(1000byte를 전송하는데 요구되는 시간(초)) 혹은 이것들과 다른 인자들의 결합을 나타낸다면 어떤 하나의 콤퓨터로부터 다른 여러 콤퓨터에로 전자신문을 보내는데 드는 비용이 최소로 되는 경로를 찾는데 최단경로알고리듬을 리용할수 있다.

우리는 항공체계 또는 다른 교통체계를 그라프로 모형화하여 두점사이의 최단로정을 계산하는데 최단경로알고리듬을 리용할수 있다. 많은 실천적응용에서는 하나의 정점 s로부터 오직 하나의 다른 정점 t에로의 최단경로를 구하는것이 요구되게 된다. 일반적으로 어떤 정점 s로부터 하나의 다른 정점 t에로 경로찾기가 s로부터 모든 정점에로 경로찾기보다 더 빠른 알고리듬은 없다.

이 문제에 대한 4가지 해결방안을 주는 알고리듬을 고찰하자. 먼저 무게를 가지지 않는 최단경로문제를 고찰하고 $O(|E|+|V|)$시간에 그것을 해결하는 방법을 보자. 다음에(부의 무게변을 가지지 않는다는 조건에서) 무게붙은최단경로문제를 해결하는 방법을 보자. 이 알고리듬의 실행시간은 적당한 자료구조로 실현되였다면 $O(|E|\log|V|)$이다. 그라프가 부의 무게변을 가진다면 간단한 풀이가 있지만 이것은 유감스럽게도 $O(|E|\cdot|V|)$의 불충분한 시간한계를 준다. 마지막에 우리는 비순환그라프의 특수한 경우에 대하여 무게붙은문제를 선형시간내에 풀게 될것이다.

1. 무게 없는 최단경로

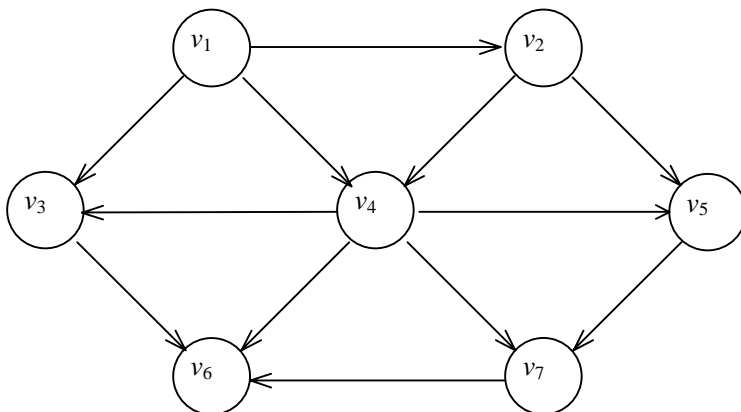



그림 9-7. 무게 없는 방향그라프 G

그림 9-7은 무게없는그라프 G를 보여 준다. 입력파라메터인 어떤 정점 s를 리용하

여 s로부터 모든 다른 정점에로 이르는 최단경로를 찾아 보자. 변들에는 무게가 없으므로 다만 경로우에 포함되는 변들의 개수에만 관심을 가진다.

이것은 명백하게 무게붙은최단경로문제의 특수한 경우이다. 그것은 모든 변들에 무게가 없으므로 1로 할당되였기때문이다.

지금은 실제경로 그자체가 아니라 최단경로길이에만 관심을 가진다고 가정하자. 실제경로를 따라가 보면서 간단한 보조적인 문제를 만들어 보자.

v_3을 s로 선택하였다고 가정하자. 인차 s로부터 v_3에로의 최단경로는 길이가 0인 경로라는것을 알수 있다. 그림 9-8의 그라프와 같이 이 정보를 표시할수 있다.

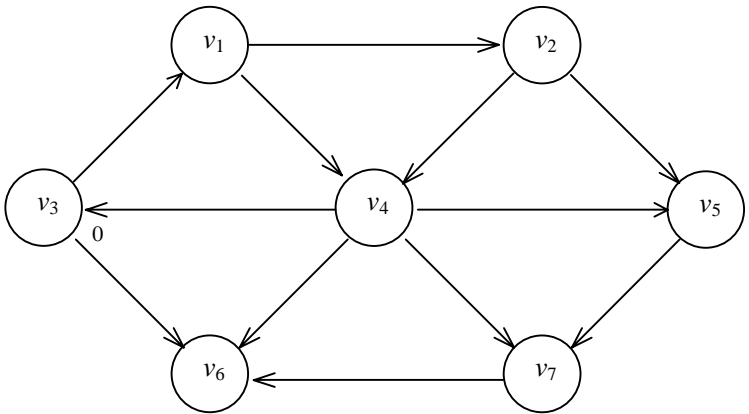



그림 9-8. 0변들에 도달 가능한 시작매듭을 표식한후의 그라프

이제 s로부터 거리가 1인 모든 정점들을 찾는것으로부터 시작한다. 이것은 s와 린접한 정점들을 찾아 봄으로써 처리할수 있다. 이렇게 하면 v_1과 v_6이 s로부터 하나의 변만큼 떨어 져 있다는것을 알수 있다. 이것은 그림 9-9에서 보여 주었다.

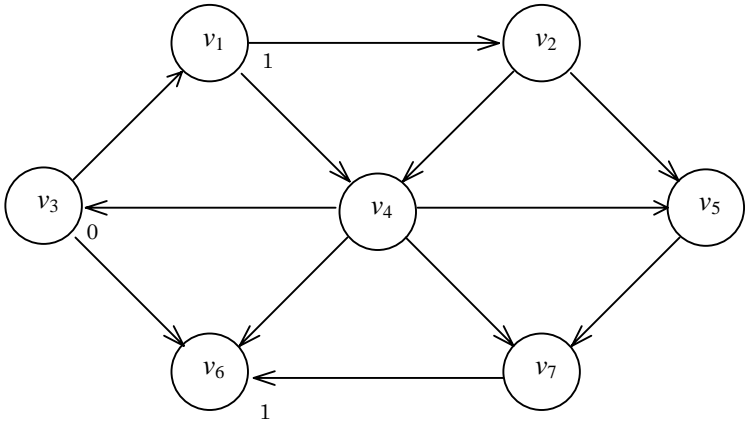



그림 9-9. s로부터 경로길이가 1인 모든 정점들을 찾은후의 그라프

최단경로가 이미 알려 져 있지 않은 v_1과 v_6에 린접한 모든 정점(거리1만큼 떨어 진 정점들)들을 찾음으로써 s로부터 최단경로가 정확히 2인 정점들을 찾을수 있다. 이 탐색결과는 v_2와 v_4에 대한 최단경로가 2이라는것을 알려 준다. 그림 9-10은 지금까지 이루어 진 진행과정을 보여 준다.

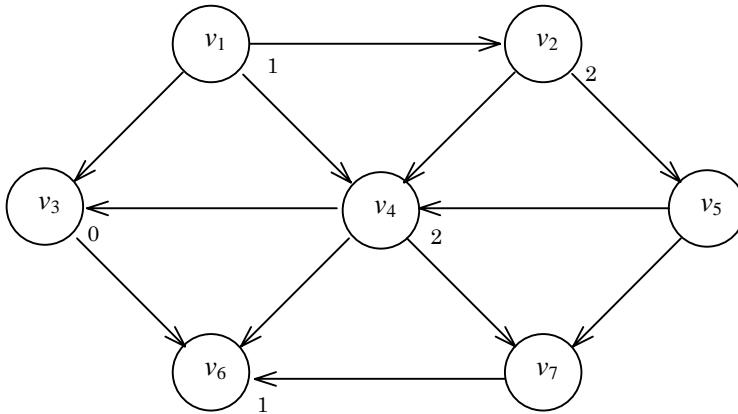



그림 9-10. 최단경로가 2인 모든 정점들을 찾은후의 그라프

마지막으로 최근에 평가된 v_2와 v_4의 린접한 정점들을 조사함으로써 v_5와 v_7이 3개의 변을 가지는 최단경로라는것을 알수 있다. 모든 정점들은 현재 모두 계산되였다. 따라서 그림 9-11은 알고리듬의 최종결과를 보여 준다.

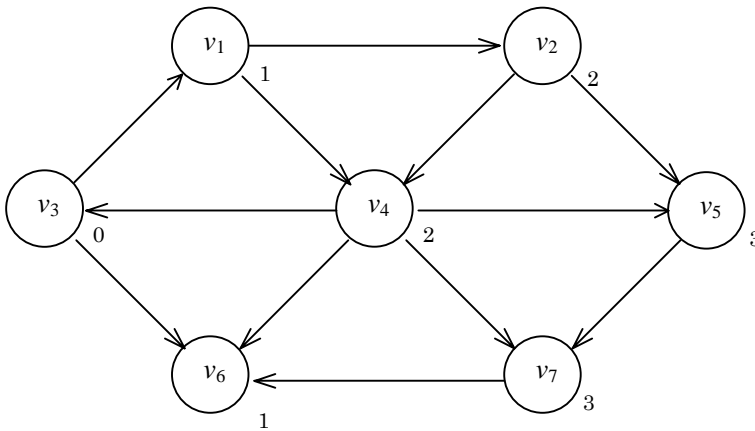



그림 9-11. 총체적인 최단경로

그라프를 탐색하기 위한 전략은 **너비우선탐색**(*breadth-first search*)으로 알려 져 있다. 이것은 층별로 정점들을 처리함으로써 조작할수 있다. 즉 시작정점에 가장 가까운 정점들이 먼저 평가되고 가장 멀리 떨어 진 정점들은 마지막에 평가된다. 이것은 나무에서 준위-순서순회와 같은것이다.

이 전략이 주어 지면 그것을 코드로 넘길수 있다. 표 9-2는 그 진행과정을 따라 가

면서 알고리듬이 리용하게 되는 표의 초기구성을 보여 준다.

매 정점에 대하여 3개 부분으로 된 정보를 따라 가면서 볼수 있다. 먼저 s로부터 거리를 입력점 d_r에 보관한다. 초기에 모든 정점들은 경로의 길이나 정점 s외에는 도달할수 없다. p_r내의 입력점은 보조변수이며 이것은 실제경로를 출력하는데 리용할수 있다. *known*입력점은 정점이 처리된 다음 true로 설정된다. 초기에 모든 입력점들은 시작정점을 포함하여 *known*이 아니다. 정점에 *known*이라는 표식기호가 붙었다면 값 눅은 경로는 더 이상 찾을수 없으므로 모든 정점에 대한 처리가 완료되였다는것을 담보할수 있다.

표 9-2. 무게 없는 최단경로계산에 리용되는 표의 초기구성

	Known	d_v,	p_v
v_1	F	∞	0
v_2	F	∞	0
v_3	F	0	0
v_4	F	∞	0
v_5	F	∞	0
v_6	F	∞	0
v_7	F	∞	0

기본알고리듬을 프로그람 9-3에 보여 주었다. 프로그람 9-3에 있는 알고리듬은 거리 d=0에서, 그다음은 d=1, d=2 등에서 정점들을 *known*으로 선언하여 거리 d_w=d+1에서 d_w=∞를 가지는 모든 린접점 w를 설정함으로써 표를 얻을수 있다.

변수 p_v를 통하여 반대로 추적하면 실제 경로를 출력할수 있다. 어떻게 하는가하는것은 무게를 가진 경우를 설명하면 알수 있다.

알고리듬의 실행시간은 2중 *for*순환이기때문에 $O(|V|^2)$이다. 비효률성은 명백히 모든 정점들이 훨씬 이전에 *known*으로 되였음에도 불구하고 NUM_VERTICES-1일 때까지 바깥순환을 계속하는것이다. 이것을 피하기 위하여 특별한 검사가 진행된다고 하여도 (입력이 시작정점 v_9를 가진 그림 9-12의 그라프인 때의 처리과정을 일반화하여 보여 주는것처럼) 최악의 경우의 실행시간에 영향을 주지 못한다.

```
            void Graph::unweighted( Vertex s )
             {
                 Vertex v, w;

/*1*/            s.dist = 0;
/*2*/            for( int currDist = 0; currDist < NUM_VERTICES; currDist++ )
/*3*/                 for each vertex v
/*4*/                      if( !v.known && v.dtst == currDist )
                           {
/*5*/                           v. known = true;
/*6*/                           for each w adjacent to v
/*7*/                                if( w.dist == INFINITY )
                                     {
/*8*/                                     w.dist = currDist + 1;
```

```
/*9*/                    w.path = v;
                       }
                   }
              }
```

프로그람 9-3. 무게없는최단경로알고리듬에 관한 가상코드

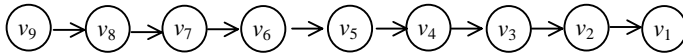



그림 9-12. 프로그람 9-3을 리용한 무게없는 최단경로알고리듬에 관한 좋지 못한 경우

위상학적정렬방법으로 이런 비효률성을 제거할수 있다. 실행시 어떤 점에서 $d_v \neq \infty$를 가지는 두가지 형태의 *unknowm*정점들만이 있다. 일부는 d_v=currDist를 가지며 나머지는 d_v=currDist+1을 가진다. 이 특별한 구조로 하여 적당한 정점을 찾기 위하여 3행과 4행에서처럼 전체 표를 탐색하는것은 아주 비효률적이다.

매우 간단하지만 추상적인 해결법은 2개의 통을 보유하는것이다. box # 1은 d_v=currDist를 가진 알려 지지 않은 정점을 가지며 box # 2는 d_v=currDist+1을 가진다. 3행과 4행에서 검사는 box # 1에서 임의의 정점을 찾음으로써 교체할수 있다. 9행뒤에 *w*를 box # 2에 첨부할수 있다. 제일 바깥의 for순환이 끝난후에 box # 1이 비게 되며 box＃2는 for순환의 다음통과에서 box # 1로 옮겨 질수 있다.

하나의 대기렬을 리용하여 이 사상을 더욱 다듬을수 있다. 첫 통과에서 대기렬은 currDist거리의 정점들만 포함한다. 거리 currDist+1의 린접정점들을 첨부하면 그것이 뒤부분에 남아 있으므로 거리 currDist의 모든 정점들이 처리된후에도 그것들은 처리되지 않는다는것을 알수 있다. 거리 currDist마지막정점이 뽑아 진후에 처리되는 대기렬은 오직 거리 currDist+1의 정점들만 포함하므로 이 처리는 영구화할수 있다. 우리는 순수 대기렬에 시작매듭을 놓아 처리를 시작할 필요가 있다.

다듬어 진 알고리듬을 프로그람 9-4에서 보여 주었다. 가상코드에서 시작정점 *s*는 파라메터로서 넘겨 진다고 가정하였다. 또한 일부 정점들이 시작매듭에 도달할수 없다면 대기렬은 때이르게 빌수 있다. 이 경우에 INFINITY거리는 이 매듭에 관하여 알려 질것이며 이것은 완전히 정당하다. 마감에 *known*자료성원은 리용되지 않는다. 정점이 처리되자마자 대기렬에 다시 들어 올수 없으므로 맹목적으로 다시 처리할 필요가 없다. 따라서 *known*자료성원은 버리게 된다. 표 9-3은 리용하고 있는 그라프상의 값들이 알고리듬처리과정에 어떻게 변화되는가를 보여 준다. *Known*자료성원을 보존함으로써 이 절의 나머지 부분과 일관성을 보장하며 표를 더 쉽게 한다.

```
Void Graph::unweighted( Vertex s )
{
      Queue q( NUM_VERTICES );
       Vertex v, w;

/* 1*/        q.enqueue( s );
/* 2*/        s.dist = 0;

/* 3*/        while( !q.isEmpty( ))
               {
/* 4*/                 v = q.dequeue( );
/* 5*/                 v.known = true;    // Not really needed anymore

/* 6*/                 for each w adjacent to v
/* 7*/                      if ( w.dist == INFINITY )
                       {
/* 8*/                           w.dist = v.dist +1;
/* 9*/                           w.path = v;
/* 10*/                          q.enqueue( w );
                       }
               }
   }
```

프로그람 9-4. 무게없는최단경로알고리듬에 대한 가상코드

위상학적정렬에서 진행한것과 같은 분석방법을 리용하여 린접표를 리용하면 그 실행시간은 $O(|E|+|V|)$이라는것을 알수 있다.

2. 딕스트라알고리듬

만일 그라프가 무게를 가진다면 문제는 더 어려워 지지만 아직까지는 무게를 가지지 않는 경우의 방법을 리용할수 있다.

앞에서와 같이 류사한 정보를 모두 보존하자. 따라서 매개 정점은 *Known* 혹은 *unKnown*중의 하나로 표식된다. 앞에서와 같이 매 정점에 대하여 검사거리 d_v를 보관한다. 이 거리는 중간결과로서 *Known*정점들만 리용하는 s로부터 v까지의 최단경로길이를 준다. 앞에서와 같이 p_v는 d_v를 변화시키는 마지막정점이다.

단일원천최단경로문제를 해결하는 일반적인 방법을 **딕스트라알고리듬**(*dijkstra's algorithm*)이라고 한다. 30년전의 이 해결방법은 탐욕알고리듬의 원시적인 실례이다. 탐욕알고리듬은 일반적으로 매 단계에서 가장 좋은 처리가 진행되도록 함으로써 단계적으로 하나의 문제를 해결한다. 실례로 미국에서 화페를 교환하기 위하여 사람들은 먼저 25센

트를 세고 그다음 10센트, 5센트, 잔돈으로 세여 내려 간다. 이 탐욕알고리듬은 쇠돈의 최소수를 리용하여 쇠돈을 교환한다. 탐욕알고리듬의 기본문제는 그것이 항상 수행되지 않는다는것이다. 12센트짜리 쇠돈의 추가는 15센트를 돌려 주는 쇠돈교환알고리듬을 파괴한다. 왜냐하면 그 결과(하나의 12센트짜리와 3개의 페니)가 최적인것이 못되기때문이다(하나의 10센트은돈과 하나의 5센트짜리 흰구리돈).

표 9-3. 무게없는최단경로알고리듬과정에 자료변화방식

v	초기상태 Known d_v, p_v	v_3이 대기렬에서 나온후 Known d_v, p_v	$v1$이 대기렬에서 나온후 Known d_v, p_v	V_6이 대기렬에서 나온후 Known d_v, p_v
v_1	F ∞ 0	F 1 0	T 1 v_3	T 1 v_3
v_2	F ∞ 0	F ∞ 0	F 2 v_1	F 2 v_1
v_3	F 0 0	T 0 0	T 0 0	T 0 0
v_4	F ∞ 0	F ∞ 0	F 2 0	F 2 v_1
v_5	F ∞ 0	F ∞ 0	F ∞ 0	F ∞ 0
v_6	F ∞ 0	F 1 v_3	F 1 v_3	T 1 v_3
v_7	F ∞ 0	F ∞ 0	F ∞ 0	F ∞ 0
Q	v_3	v_1, v_6	v_6, v_2, v_4	v_2, v_4

v	v_2이 대기렬에서 나온후 Known d_v, p_v	v_4가 대기렬에서 나온후 Known d_v, p_v	v_5가 대기렬에서 나온후 Known d_v, p_v	V_7이 대기렬에서 나온후 Known d_v, p_v
v_1	T 1 v_3	T 1 v_3	T 1 v_3	T 1 v_3
v_2	T 2 v_1	T 2 v_1	T 2 v_1	T 2 v_1
v_3	T 0 0	T 0 v_2	T 0 0	T 0 0
v_4	F 2 v_1	T 2 v_1	T 2 v_1	T 2 v_1
v_5	F 3 v_2	F 3 0	T 3 v_2	T ∞ v_2
v_6	T 2 v_3	T 1 v_3	T 1 v_3	T 1 v_3
v_7	F ∞ 0	F 3 v_4	F 3 v_4	T ∞ v_4
Q	v_4, v_5	v_5, v_7	v_7	빈상태

딕스트라알고리듬은 무게없는최단경로알고리듬들과 같이 단계적으로 처리된다. 매 단계에서 딕스트라알고리듬은 *unKnown*인 모든 정점들가운데서 가장 작은 검사거리 d_v를 가지는 정점 v를 선택하고 s로부터 v에로 최단경로를 *Known*으로 선언한다. 나머지단계는 d_w의 값을 갱신한다. 무게를 가지지 않는 경우에 $d_w=\infty$라면 $d_w=d_v+1$로 설정한다. 따라서 정점 v가 보다 짧은 경로를 준다면 d_w의 값을 더 낮춘다. 무게를 가진 경우에 같은 론리를 적용하면 d_w에 대한 이 새로운 값으로 개선되는 경우 $d_w=d_v+c_{vw}$로 설정한다. 간단히 말하여 알고리듬은 w에로의 경로우에서 v를 리용하는것이 좋은가 나쁜가 하는것을 판정

한다. 초기값 d_w는 v를 리용하지 않는 경우의 값이며 우에서 계산된 값은 v와 *Known*정점들만을 리용하는 눅거리경로이다.

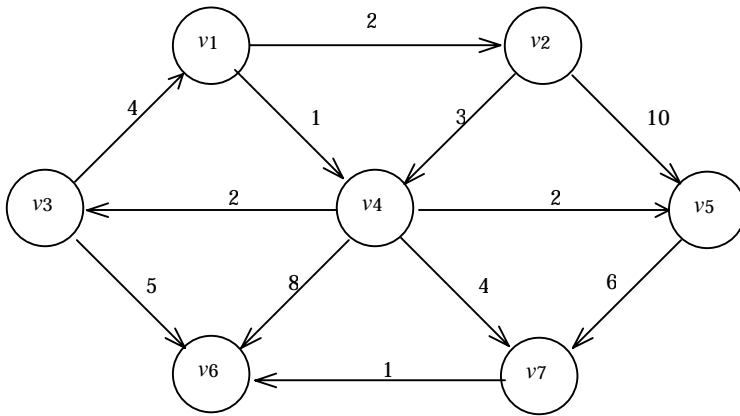



그림 9-13. 방향그라프 G(다시한번)

그림 9-13에서 보여 준 그라프가 그의 실례이다. 표 9-4는 시작매듭 s가 v_1이라고 가정했을 때의 초기구성을 나타낸다. 선택된 첫 정점은 v_1이며 길이가 0인 경로를 가진다. 이 정점에는 *Known*라는 표식기호를 붙인다. v_1이 *Known*이면 일부 입력점들은 조절되여야 한다. v_1과 린접인 정점들은 v_2와 v_4이다. 표 9-5에서 보여 주는바와 같이 이 정점들은 둘다 조정된 입력점들이다.

표 9-4. 딕스트라알고리듬에서 리용된 초기구성표

v	*Known*	d_v,	p_v
v_1	F	0	0
v_2	F	∞	0
v_3	F	∞	0
v_4	F	∞	0
v_5	F	∞	0
v_6	F	∞	0
v_7	F	∞	0

표 9-5. v_1가 *Known*으로 선언된후

v	*Known*	d_v,	p_v
v_1	T	0	0
v_2	F	2	v_1
v_3	F	∞	0
v_4	F	1	v_1
v_5	F	∞	0
v_6	F	∞	0
v_7	F	∞	0

다음 v_4가 선택되고 거기에 *Known*이라는 표식기호를 붙인다. 정점 v_3, v_5, v_6, v_7들은 v_4의 린접이며 표 9-6에서 보여 주는바와 같이 모두가 조정을 요구하게 되여 있다.

다음에 v_2이 선택된다. v_4는 린접이지만 이미 *Known*으로 되였으므로 그에 대하여 그 어떤 처리도 수행되지 않는다. v_5는 린접이지만 조정되지 않는다. 그것은 v_2를 통과하는 값이 2+10=12이고 길이가 3인 경로가 이미 알려 져 있기때문이다. 표 9-7은 이 정점들이 선택된후의 표를 보여 준다.

표 9-6. v_4가 *Known*으로 선택된후

v	Known	d_v	p_v
v_1	T	0	(
v_2	F	2	v
v_3	F	3	v
v_4	T	1	v
v_5	F	3	v
v_6	F	9	v
v_7	F	5	v

표 9-7. v_2이 *Known*으로 선택된후

v	Known	d_v	p_v
v_1	T	0	0
v_2	T	2	v
v_3	F	3	v
v_4	T	1	v
v_5	F	3	v
v_6	F	9	v
v_7	F	5	v

선택된 다음 정점은 값이 3인 v_5이다. v_7은 유일한 린접점이지만 조정되지 않는다. 그것은 3+6>5이기때문이다. 그다음 v_3이 선택되고 v_6에 관한 거리는 3+5=8로 맞추어 진다. 그 결과표를 표 9-8에 보여 준다.

다음 v_7이 선택되고 v_6은 5+1=6으로 갱신된다. 그 결과표를 표 9-9에 보여 주었다.

표 9-8. v_5와 그다음 v_3이 *Known*으로 선택된후

v	Known	d_v	p_v
v_1	T	0	(
v_2	T	2	v
v_3	T	3	v
v_4	T	1	v
v_5	T	3	v
v_6	F	8	v
v_7	F	5	v

표 9-9. v_7이 *Known*으로 선택된후

v	Known	d_v	p_v
v_1	T	0	0
v_2	T	2	v_1
v_3	T	3	v_4
v_4	T	1	v_1
v_5	T	3	v_4
v_6	F	6	v_7
v_7	T	5	v_4

마감에 v_6이 선택된다. 최종표를 표 9-10에 주었다. 그림 9-14는 딕스트라알고리듬실행과정에 변들이 어떻게 *Known*으로 표식되며 정점들이 갱신되는가를 도식적으로 보여 준다.

표 9-10. 딕스트라알고리듬의 단계들

v	Known	d_v	p_v
v_1	T	0	0
v_2	T	2	v_1
v_3	T	3	v_4
v_4	T	1	v_1
v_5	T	3	v_4
v_6	T	6	v_7
v_7	T	5	v_4

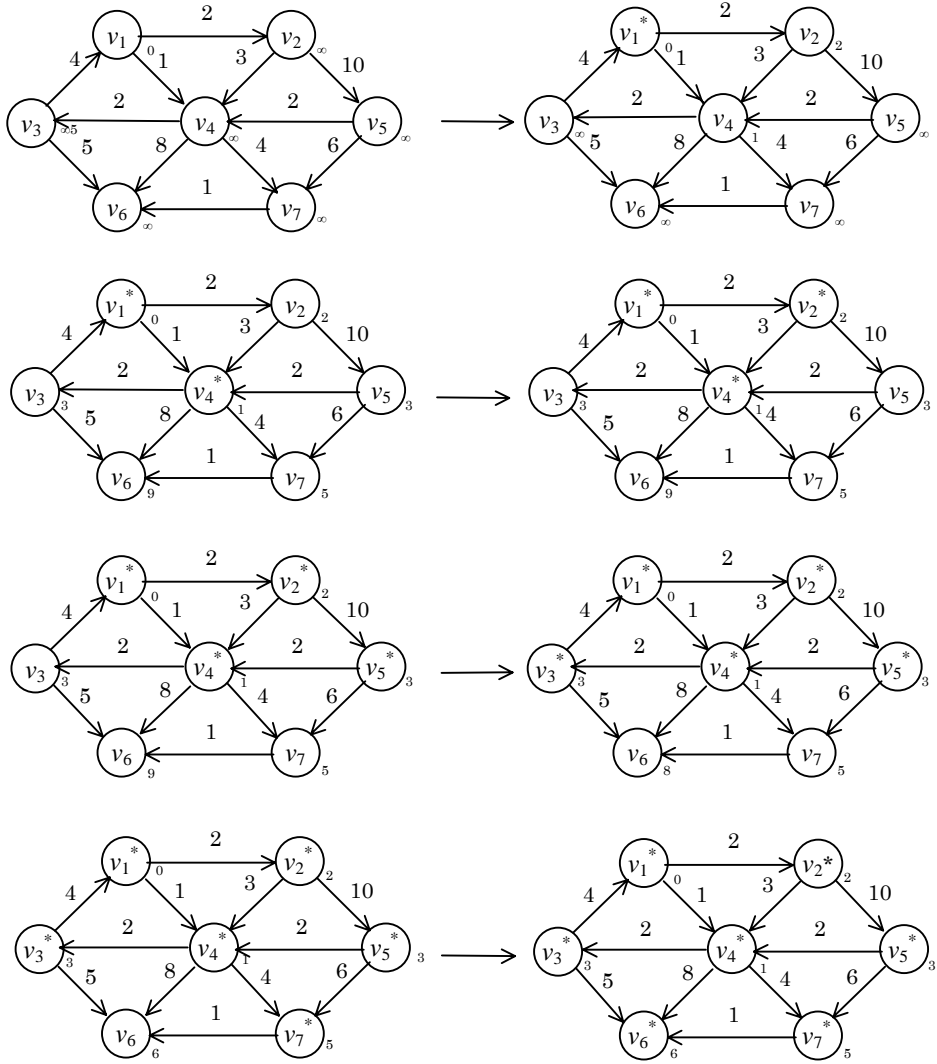



그림 9-14. 딕스트라알고리듬의 단계들

시작정점으로부터 어떤 정점 *v*에로 실제경로를 출력하기 위하여 *p*개 변수들에서 왼쪽 끝에 따르는 재귀루틴을 쓸수 있다.

이제 딕스트라알고리듬을 실현하기 위한 가상코드를 보기로 하자. 매 Vertex는 알고리듬에서 리용되는 여러가지 자료성원들을 보관한다. 이것을 프로그람 9-5에 보여 주었다. 그라프는 readGragph루틴에 의하여 모두 린접표로 구축되여 Vertex의 배렬에로 읽어들인다고 가정하자. 프로그람 9-6에서 보여 주는바와 같이 다른 자료성원들을 초기화하는것은 간단한 문제이다.

그 경로는 프로그람 9-7에서 준 재귀루틴을 리용하여 출력할수 있다. 이 루틴은 경

로우에서 v앞의 정점까지의 경로를 모두 재귀적으로 출력하고 그다음 v를 출력한다. 그 경로가 단순하므로 이 루틴은 쉽게 처리된다.

```
/.**
  *   PSEUDOCODE sketch of the Vertex structure.
  *   In real C++, path would be of type Vertex *,
  *   and many of the code fragments that we describe
  *   require either a dereferencing * or use the
  *   operator instead of the . operator.
  *   Needless to say, this obscures the basic algorithmic
  *   ideas. The Appendix and online code have an example
  *   of working C++ code.
  */
struct Vertex
{
    List          adj;        // Adjacency list
    bool          known;
    DistType dist;            // DistType is probably int
    Vertex        path;       // Probably Vertex *, as mentioned above
                              // Other data and member functions as needed };
}
```

프로그람 9-5. 딕스트라알고리듬에 대한 Vertex클라스

```
            void Graph::createTab1e( vector<Vertex> & t )
             {
/*1*/           readGraph( t );      // Read graph somehow; fi11 in adj
/*2*/           for( int i = 0; i < t.size( ); i++ )
                {
/*3*/              t[ i ] .known = false;
/*4*/              t[ i ].dist = INFINITY;
/*5*/              t[ i ].Lpath = NOT_A_VERTEX;    // NOT_A_VERTEX is probably NULL
                }
/*6*/           NUM_VERTICES = t.size( );
             }
```

프로그람 9-6. Vertex배렬을 되돌리는 루틴

프로그람 9-8은 기본알고리듬을 보여 주는데 이것은 탐욕선택규칙을 리용하여 표를 만드는 for순환고리를 포함한다.

부정에 의한 증명방법으로 이 알고리듬이 그 어느 변도 부의 무게를 가지지 않는 조건에서 항상 동작한다는것을 증명할수 있다. 어떤 변이 부의 무게를 가진다면 알고리듬은 잘못된 결과를 내보낸다(련습문제 9-7 ㄱ를 보시오.). 실행시간은 정점을 다루는 방

법에 관계되는데 이것은 앞으로 고찰하려고 한다. 최소 d_v를 찾기 위하여 정점들의 배렬을 아래로 주사하는 명백한 알고리듬을 리용한다면 매 단계는 최소값을 찾는데 $O(|V|)$시간이 걸리며 따라서 알고리듬과정에서 최소값을 찾는데는 $O(|V|^2)$시간이 소비된다. d_w를 갱신하는 시간은 갱신건당 상수적이며 전체 $O(|E|)$에 대하여 변당 많아서 한번은 갱신이 있다. 따라서 전체 실행시간은 $O(|E|+|V|^2)= O(|V|^2)$으로 된다.

```
/**..
* Print shortest path to v after dijkstra has run.
*   Assume that the path exists.
*/
void Graph: :printPath( Vertex v )
{
      if( v.path != NOTJLVERTEX )
      {
            printPath( v.path );
            cout << " to ";
      }
        cout << v;
}
```

프로그람 9-7. 실제적인 최단경로를
출력하는 루틴

```
            void Graph::dijkstra( Vertex   s )
            {
            Vertex v, w;
 /*1*/           s.dist = 0;
 /*2*/           for( ; ; )
                 {
 /*3*/               v = smallest unknown distance vertex;
 /*4*/               if( v = = NOT_A_VERTEX )
 /*5*/                  break;
 /*6*/                  v.known = true;
 /*7*/                  for each w adjacent to v
 /*8*/                  if( !w.known )
 /*9*/                  if( v.dist + cvw < w.dist )
                          {        // Update w
 /*10*/                       decrease( w.dist to v.dist + cvw );
 /*11*/                       w.path = v;
                          }
                   }
              }
```

프로그람 9-8. 딕스트라알고리듬에 대한 가상코드

그라프가 $|E|=\Theta(|V|^2)$로서 조밀하면 이 알고리듬은 간단할뿐아니라 본질적으로 최적이다. 그것은 변의 수에 따라 선형시간에 실행되기때문이다. 그라프가 $|E|=\Theta(|V|)$로서 성글다면 이 알고리듬은 지내 속도가 느리다. 이 경우에 거리를 우선권대기렬에 보관할 필요가 있다. 이렇게 하는데 실제로 두가지 방안이 있는데 둘 다 류사하다.

3행과 6행은 deleteMin연산을 수행하기 위하여 결합한다. 왜냐하면 알려 지지 않은 최소정점이 일단 발견되면 그것은 더이상 미지수가 아니며 그다음의 고찰에서는 제거되여야 하기때문이다.

한가지 방안은 decreasekey연산과 류사하게 처리를 갱신한다. 최소값을 찾는 시간은 $O(\log|V|)$이다. 그것은 decreasekey연산과 마찬가지로 갱신을 진행하는 시간이다. 이것은 $O(|E|\log|V|+|V|\log|V|)=O(|E|\log|V|)$의 실행시간을 주므로 이전의 성긴 그라프들에 한해서는 개선으로 된다. 우선권대기렬은 find연산을 효률적으로 지원하지 못하므로 우선권대기렬에서 d_i의 매개 값의 위치는 d_i가 변할 때마다 보존되고 갱신될 필요가 있다. 우선권대기렬이 2진더미에 의하여 실현된다면 이것은 번거러워 지며 쌍더미(제12장)가 리용되면 코드는 더욱 나빠진다.

다른 방법은 10행이 실행될 때마다 우선권대기렬에 w와 새로운 값 d_w를 삽입하는것이다. 따라서 우선권대기렬내의 매개 정점에 대하여 여러가지로 표현될수 있다. deleteMin연산이 우선권대기렬에서 최소정점을 제거하면 그것이 이미 *known*으로 되여 있지 않는가를 검사해야 한다. 따라서 *unknown*정점이 나타날 때까지 3행에서 deleteMin연산이 반복되게 된다. 이 방법은 쏘프트웨어관점에서 보면 우수하고 코드화하기가 확실히 매우 쉽지만 우선권대기렬의 크기는 $|E|$만큼 커지게 된다. 이것은 $|E|\leqq|V|^2$가 $\log|E|\leqq 2\log|V|$라는것을 암시하므로 접근시간한계에 영향을 주지 못한다. 따라서 여전히 $O(|E|\log|V|$알고리듬들을 가지게 된다. 그러나 기억기요구는 증가하므로 일부 응용에서 중요할수 있다. 게다가 이 방법은 오직 $|V|$대신에 $|E|$번의 deleteMin연산을 요구하기때문에 실천적으로 속도가 더 느릴수 있다.

콤퓨터전자우편이나 수화물운송과 같은 전형적인 문제들에서 대부분의 정점들은 2~3개의 변들만 가지기때문에 그라프는 매우 성글므로 많은 응용들에서 이 문제를 해결하는데 우선권대기렬을 리용하는것이 좋다.

각이한 자료구조를 리용하면 딕스트라알고리듬을 리용하여 보다 좋은 시간한계를 얻을수 있다. 제11장에서 피보나치더미라고 하는 또 다른 우선권대기렬자료구조를 고찰하게 된다. 이것을 리용하면 그 실행시간은 $O(|E|+|V|log|V|)$이다. 피보나치더미들은 좋은 리론적인 시간한계를 가지지만 약간한 자리넘침을 가진다. 따라서 피보나치더미가 2진더미를 가진 딕스트라알고리듬보다 실천적으로 더 좋은지는 명백하지 않다. 현재까지 이 문제에 대하여 의의 있는 평균경우의 결과들은 없다.

3. 부의 무게를 가지는 그라프

그라프가 부의 무게값을 가진다면 딕스트라알고리듬은 처리되지 않는다. 문제는 정점 *u*가 *known*으로 선언되면 어떤 다른 *unknown*정점 *v*로부터 완전히 부인 *u*에로 다시 되돌아 오는 경로가 있을 가능성이 있다는것이다. 이러한 경우에 *s*로부터 *v*에로, 다시 *u*에로의 경로를 가지는것이 *v*를 리용하지 않고 *s*로부터 *v*에로 가는것보다 더 좋다. 련습문제 9-7의 ㄱ는 명백한 실례를 들것을 요구한다.

좋은 해답은 부의 무게변을 제거하고 매 변의 값에 상수 △를 추가하여 새로운 그라프에 대한 최단경로를 계산한 다음 본래문제에 대한 결과로 리용하는것이다. 이 전략을 그대로 실현하면 동작하지 않는다. 그것은 많은 변을 가진 경로가 몇개의 변들을 가진 경로보다 무게가 더 크기때문이다.

무게가 있는것과 없는 그라프에 대한 알고리듬들의 결합으로 이 문제를 해결할수 있지만 실행시간이 극도로 증가하게 된다. 여기서는 *known* 정점들의 개념을 없애버렸으므로 알고리듬을 변화시킬 필요가 있다. 먼저 대기렬에 정점 *s*를 넣는것으로부터 시작한다. 그다음 매 단계에서 정점 *v*를 대기렬에 넣는다. $d_w > d_v + c_{v,w}$인 *v*와 린접인 모든 정점 *w*를 찾는다. 그리고 d_w와 p_w를 갱신한 다음 *w*가 대기렬에 이미 들어 가 있지 않다면 우선권대기렬에 *w*를 넣는다. 매 정점은 그 존재를 나타내도록 대기렬에 잠간 설정할수 있다. 이 과정을 대기렬이 빌 때까지 반복한다. 프로그람 9-9는 이 알고리듬의 실현을 보여 준다.

```
              Void Graph::weightedNegative( Vertex s )
              {.
                     Queue q( NUM_VERTICES );
                     Vertex v, w;
/*1*/                       q.enqueue( s );
/*2*/                       s.dist = 0;
/*3*/                       whi1e( !q.isEmpty( ) )
                                 {
/*4*/                                  v = q.dequeue( );
/*5*/                                  for each w adjacent to v
/*6*/                                         if( v.dist + cvw < w.dist )
                                                 {
                                                        // Update w
/*7*/                                                w.dist = v.dist + cvw;
/*8*/                                                w.path = v;
/*9*/                                                if( w is not already in q )
/*10*/                                                      q.enqueue( w );
                                                }
                                   }
              }
```

프로그람 9-9. 부값을 가진 최단경로알고리듬에 관한 가상코드

알고리듬이 비록 부값순환이 없는 조건에서 동작한다고 해도 6~10행코드가 변마다 한번 실행된다는것은 사실과 맞지 않는다. 매 정점은 많아서 $|V|$번 뽑을수 있으므로 실행시간은 린접표를 리용하는 경우에 $O(|E|\cdot|V|)$으로 된다(련습문제 9-7 ㄴ). 이것은 딕스트라알고리듬을 리용하면 완전히 증가하지만 실천적인 문제들에서는 다행이도 값이 부가 아니다. 부값순환이 존재하면 씌여 진 알고리듬은 무한순환한다. 임의의 정점을 $|V|$+1번 뽑아 낸 다음 알고리듬이 정지됨으로써 완료된다.

4. 비순환그라프

그라프가 비순환이라는것을 알게 되든가 아니면 정점선택규칙을 알게 되면 정점들을 *known*으로 선언되도록 순서를 변화시켜 딕스트라알고리듬을 개선할수 있다. 새로운 규칙은 위상학적순서로 정점들을 선택하는것이다. 알고리듬은 위상학적정렬이 수행됨과 함께 선택과 갱신이 이루어 지므로 한통과에 수행할수 있다. 정점 V가 선택될 때 위상학적순서규칙에 의하여 *unknown*매듭으로부터 나오는 변들은 포함시킬수 없으며 따라서 그들의 거리 d_v는 더 낮아 질수 없게 되여 이 선택규칙이 적용된다.

이 선택규칙을 가진 우선권대기렬을 쓸 필요는 없는데 선택처리가 상수적인 시간을 가지기때문에 그 실행시간은 $O(|E|+|V|)$이다.

비순환그라프는 내리지치기스키문제(점 ㄱ로부터 ㄴ까지 오직 내려 갈수만 있으므로 명백히 순환은 없다.)로 모형화할수 있다. 다른 가능한 응용(거꾸로 할수 없는)은 화학반응의 모형화이다. 여기에서 매개 정점은 어떤 실험상태를 나타낸다. 변은 한 상태로부터 다른 상태에로 이동을 나타낸다. 그리고 변의 무게는 방출되는 에네르기를 나타낼수 있다. 보다 높은 에네르기상태로부터 보다 낮은 에네르기상태에로 이동만 허용된다면 그라프는 비순환적이다.

비순환그라프는 보다 중요하게 **림계경로분석**(*critical path analysis*)에 리용된다. 그림 9-15에 있는 그라프는 그 실례로 된다. 매 매듭은 진행해야 할 활동과 그것을 완료하는데 걸리는 시간을 나타낸다. 따라서 이러한 그라프를 **능동매듭**(*activity-node*)그라프라고 한다. 여기에서 변들은 우선권관계를 나타낸다. 즉 변 (v, w)는 활동 w가 시작되기전에 활동 v가 완료되여야 한다는것을 의미한다. 물론 이것은 그라프가 비순환이여야 한다는것을 암시한다.

서로 관계(직접적이든지 혹은 간접적이든지)되지 않는 임의의 활동이 서로 다른 봉사자들에 의하여 병렬로 수행될수 있다고 가정하자. 이러한 형태의 그라프는 대상과제를 모형화하는데 리용될수 있다. 이 경우에 관심사로 되는 여러가지 중요한 문제가 있다. 첫째로 그 대상과제를 가장 빨리 완성할수 있는 시간은 얼마인가?

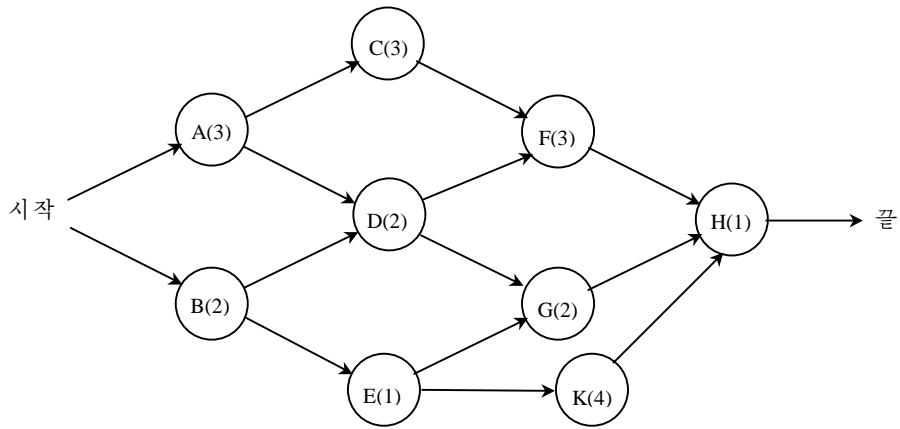



그림 9-15. 능동매듭그라프

그라프로부터 경로 *A, C, F, H*에 따라서 10시간 요구된다는것을 알수 있다. 다른 중요한 문제는 어느 활동이 지연될수 있는가 그리고 최소완성시간에 영향을 주지 않고 얼마나 오래 지연되는가를 결정하는것이다.

실례로 *A, C, F, H*중에서 임의의 지연이 발생하면 총체적인 시간은 10시간을 지나 완성시간을 초과한다. 다른 한편 활동 *B*가 덜 중요하므로 최종완성시간에 영향을 주지 않고 2시간까지 지연될수 있다.

이것들을 계산하기 위하여 능동매듭그라프를 **사건매듭**(*event-node*)그라프로 변환한다. 매 사건은 어떤 활동과 그와 관계되는 모든 활동의 완성에 대응된다. 사건매듭그라프에서 매듭 *v*로부터 도달할수 있는 사건들은 사건 *v*가 완료될 때까지 시작되지 못할수도 있다. 이 그라프는 수동 혹은 자동적으로 구성될수 있다. 활동이 여러가지 다른것들에 관계되는 경우 그 위치에 가상변들과 매듭들을 삽입할 필요가 있다. 이것은 거짓의존성이 발생하는것을 방지하기 위하여 필요하다. 그림 9-15의 그라프에 대응한 사건매듭그라프를 그림 9-16에서 보여 주었다.

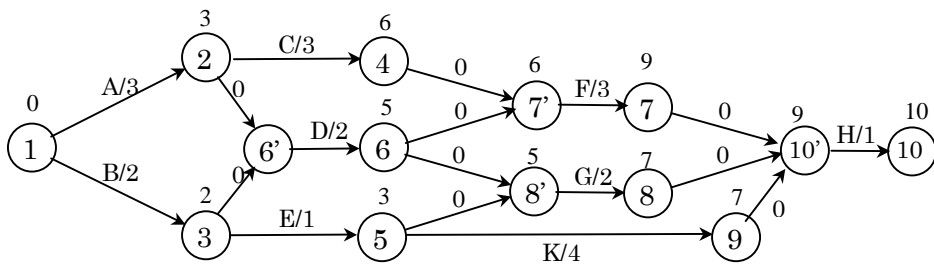



그림 9-16. 사건매듭그라프

대상과제를 가장 빨리 완성하는 시간을 찾기 위하여 단지 첫 사건으로부터 마지막 사건까지 최장경로길이를 찾아야 한다. 일반그라프에서 최장경로문제는 일반적으로 정값무게순환가능성때문에 사건에 맞지 않는다. 최단경로문제는 부값무게순환과 동등하다. **정값무게순환경로**(*positive-cost cycle*)가 존재하면 가장 긴 단순경로를 요구할수 있지만 이 문제에 대하여 만족할만한 해결책이 알려 져 있지 않다. 사건매듭그라프는 비순환이므로 순환에 대하여 걱정할 필요는 없다.

이 경우에 그라프의 모든 매듭들에 대하여 가장 빠른 완성시간을 계산하는데 최단경로알고리듬을 리용하는것이 좋다. EC_i가 매듭 i에 대한 가장 빠른 완성시간이라면 다음과 같은 규칙들을 리용할수 있다.

$$EC_1 = 0$$

$$EC_w = \max_{(v,w)\in E}(EC_v + c_{v,w})$$

그림 9-17은 사건매듭그라프에서 매 사건에 대한 가장 빠른 완성시간을 보여 준다. 또한 가장 늦은 시간 LC_i를 계산할수 있다. 매 사건은 최종완성시간에 영향을 줌이 없이 끝날수 있다. 이를 위한 식은 다음과 같다.

$$LC_n = EC_n$$

$$LC_v = \min_{(v,w)\in E}(LC_w - c_{v,w})$$

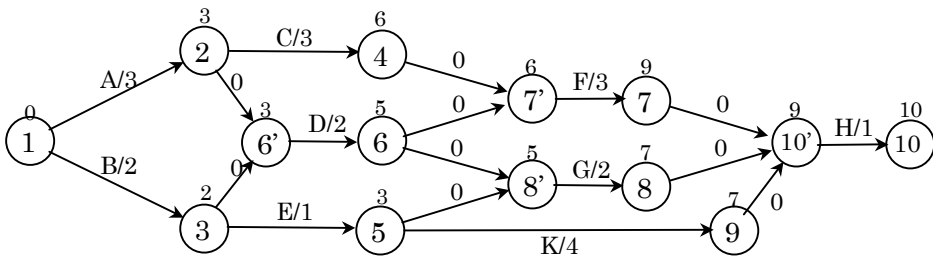



그림 9-17. 가장 빠른 완성시간

이 값들은 매 정점에 대하여 모든 린접정점과 앞선정점들의 목록을 관리함으로써 선형시간내에 계산될수 있다. 가장 빠른 완성시간은 정점들에 대하여 위상학적순서에 따라 계산되며 가장 늦은 완성시간은 위상학적역순서에 따라 계산된다. 가장 늦은 완성시간을 그림 9-18에서 보여 주었다.

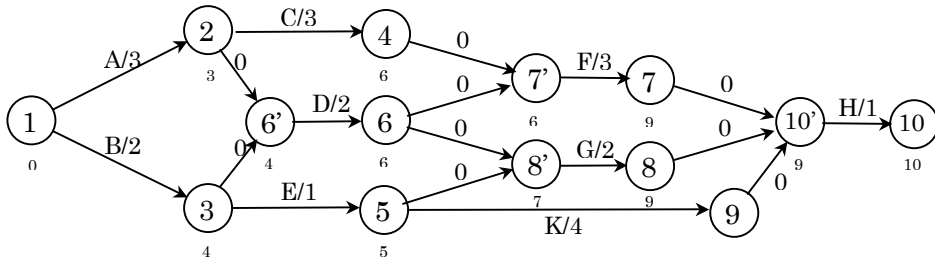



그림 9-18. 가장 늦은 완성시간

사건매듭그라프에서 매 변에 대한 느린 시간은 대응되는 활동에 대한 완료가 전체적인 완료에 지연을 줌이 없이 지연될수 있는 총 시간을 나타낸다. 그것은 다음과 같이 쉽게 고찰할수 있다.

$$Slack_{(v,w)} = LC_w - EC_v - c_{v,w}$$

그림 9-19는 사건매듭그라프에서 매개 활동에 대한 완화정도를 보여 준다. 매개 매듭에 관하여 우에 놓인 수자는 가장 빠른 완성시간이며 아래에 놓인 수자는 가장 늦은 완성시간이다.

어떤 활동들은 0 완화성을 가진다. 이것들은 림계활동들이며 일정표에서 없애야 한다. 전체적으로 완화정도가 0인 변들로 이루어 진 경로가 적어도 하나 있다. 이러한 경로가 **림계경로**(*critical path*)이다.

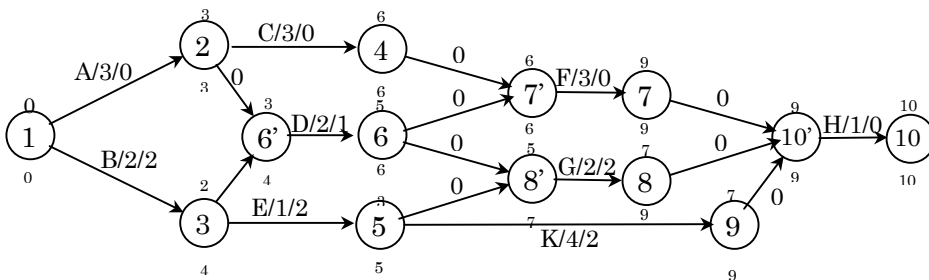



그림 9-19. 가장 빠른 완성시간, 가장 늦은 완성시간 그리고 완화정도

5. 모든 쌍들사이의 최단경로.

때때로 그라프내의 모든 정점들사이의 최단경로를 찾는것이 중요하다.

적당한 단일원천알고리듬이 $|V|$시간내에 실행되였다고 하여도 제때에 모든 정보를 계산한다면 특별히 조밀한 그라프에 대하여 보다 빠른 해결방법을 기대할수 있다.

제10장에서 무게붙은그라프에서 이 문제를 해결하는 $O(|V|^3)$알고리듬을 볼수 있다. 조밀한 그라프에 대해서는 비록 간단한(비우선권대기렬) 딕스트라알고리듬을 $|V|$시간내에

실행할 때 같은 한계를 가진다 하더라도 순환이 너무도 간결하여 모든 정점쌍들에 대한 알고리듬은 실행속도가 실천적으로 더 빠르다는것이다. 성긴그라프에 대해서는 물론 우선권대기렬로 코드화한 딕스트라알고리듬을 |V|시간에 실행하는것이 더 빨라 진다.

제4절. 망흐름문제

변의 능력이 $c_{v,w}$인 방향그라프 $G=(V, E)$가 주어 졌다고 가정하자. 이 능력은 두 교차점사이에 있는 도로우로 달리는 운반수단들의 통과량이나 관으로 흐르는 물량을 나타낸다. 2개의 정점을 가지는데 s를 **원천지**(*source*), t를 **목적지**(*sink*)라고 한다. 많아서 $c_{v,w}$의《흐름》단위가 어떤 변(v, w)로 통과한다. s도 t도 아닌 임의의 정점 v에서 들어 오는 전체 흐름은 흘러 나가는 전체 흐름과 같아야 한다. 최대흐름문제는 s로부터 t로 통과하는 최대흐름량을 결정하는것이다. 실례로 그림 9-20에 있는 그라프에서 왼쪽 그라프의 최대흐름은 오른쪽 그라프에서 보여 주는것처럼 5이다.

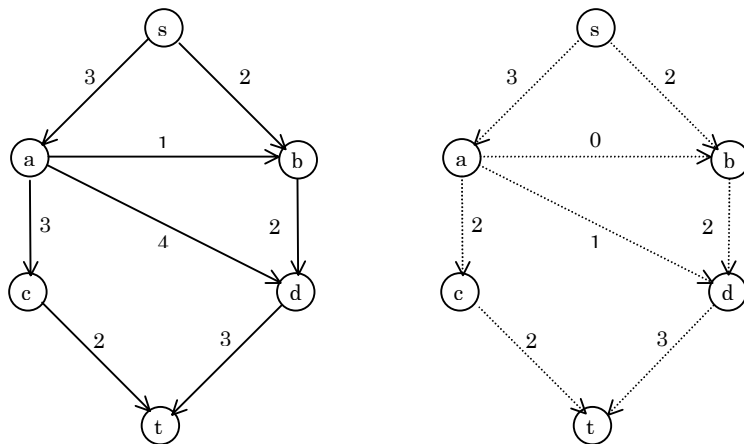



그림 9-20. 그라프(왼쪽)와 그 최대흐름

문제상태가 요구하는바와 같이 그 어느 변도 그 능력보다 많은 흐름을 통과시키지 못한다. 정점 a는 들어 오는 량이 3이고 그것을 c와 d로 분배한다.

정점 d는 a와 b로부터 입력흐름량이 3이고 이것을 결합하여 그 결과를 t에 보낸다. 정점은 변의 통과능력이 보장되는한 그리고 흐름이 유지되는한 이와 같은 방법으로 흐름을 결합, 분배할수 있다(입력되는것만큼 출력되여야 한다.).

1. 간단한 최대흐름알고리듬

이 문제를 해결하기 위한 첫 시도는 단계적으로 처리된다. 그라프 G로부터 시작하

여 흐름그라프 G_f를 구성한다. G_f는 알고리듬에서 어떤 단계에서 통과하는 흐름을 표시한다. 초기에 G_f내의 모든 변들은 흐름을 가지지 않으며 그 알고리듬이 끝나면 G_f는 최대흐름을 포함하게 된다. 또한 **나머지그라프**(*residual*)라고 하는 *Gr*그라프를 구성한다. *Gr*는 매 변에 대하여 얼마나 많은 흐름이 첨부되는가를 알려 준다. 매 변에 대한 통과능력에서 현재 흐름을 덜어 냄으로써 이 량을 계산할수 있다. *Gr*의 변을 **나머지변**(*residual*)이라고 한다.

매 단계에서 *s*로부터 *t*에로 *Gr*내의 경로를 찾는다. 이 경로를 **증가경로**(*augmenting path*)라고 한다. 이 경로상의 최소변은 경로상의 모든 변에 첨부될수 있는 흐름량이다. G_f를 조정하고 *Gr*를 다시 계산하여 이러한 처리를 수행할수 있다. *Gr*에서 s로부터 *t*에로 경로를 찾지 못한 경우 끝낸다. 이 알고리듬은 *s*로부터 *t*에로 임의의 경로를 선택할수 있다는데로부터 비결정적인데 명백하게 이보다 더 좋은 다른 선택안이 있다. 이에 대하여 후에 고찰하게 된다. 우의 실례를 가지고 이 알고리듬을 실행시켜 보자. 아래에 있는 그라프는 *G*, G_f, *Gr*이다. 이 알고리듬에 약간한 결함이 있다. 초기구성을 그림 9-21에 보여 주었다.

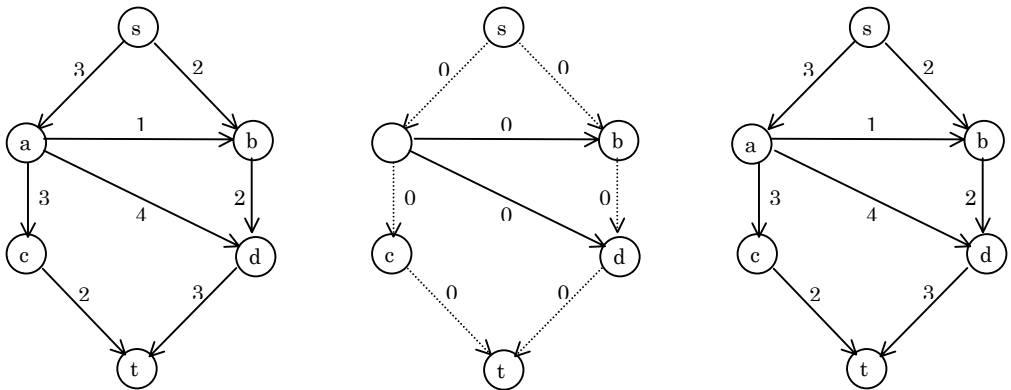



그림 9-21. 그라프의 초기상태, 흐름그라프, 나머지그라프

나머지그라프에는 *s*에서 *t*까지 많은 경로가 있다. *s, t, d, t*를 선택하였다고 하자. 그때 이 경로상에 모든 변을 통하여 두개의 흐름단위를 보낼수 있다. 어떤 변의 통과량이 다 차면 즉시 나머지그라프에서 그것을 제거한다는 규칙을 적용하자. 그러면 그림 9-22와 같은 상태를 얻을수 있다.

다음에 경로 *s, a, c, t*를 선택하는데 이것은 두개의 흐름단위를 통과시킬수 있다. 요구된 조정결과를 그림 9-23의 그라프로 보여 주었다. 선택하여야 할 경로는 오직 하나인데 그것은 *s, a, d, t*로서 하나의 흐름단위를 통과시킬수 있다. 이 결과적인 그라프를 그림 9-24에서 보여 주었다. 이 시점에서 *t*가 *s*로부터 도달할수 없기때문에 알고리듬이 끝난다. 결과 흐름 5가 최대값으로 된다. 문제점들을 고찰하기 위하여 초기그라프에서 경로 *s, a,*

d, *t*를 선택하였다고 가정하자.

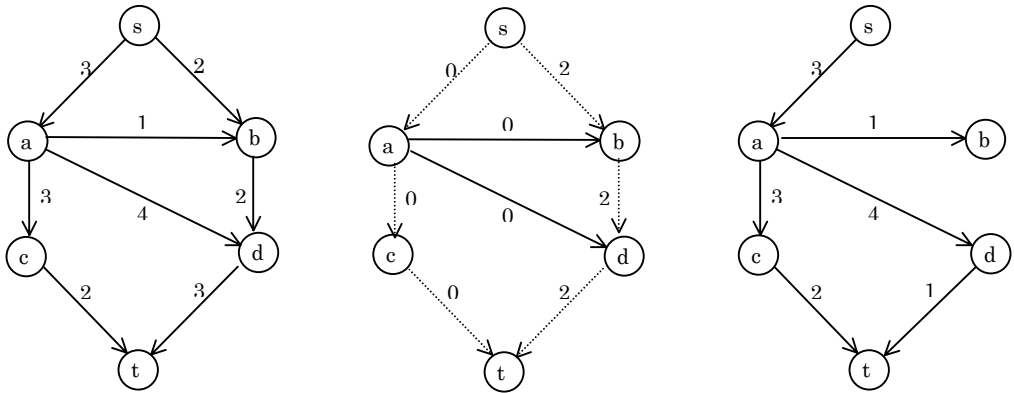



그림 9-22. 두개의 흐름단위가 *s*, *b*, *d*, *t*에 따라 추가된후의 *G*, G_f, *Gr*

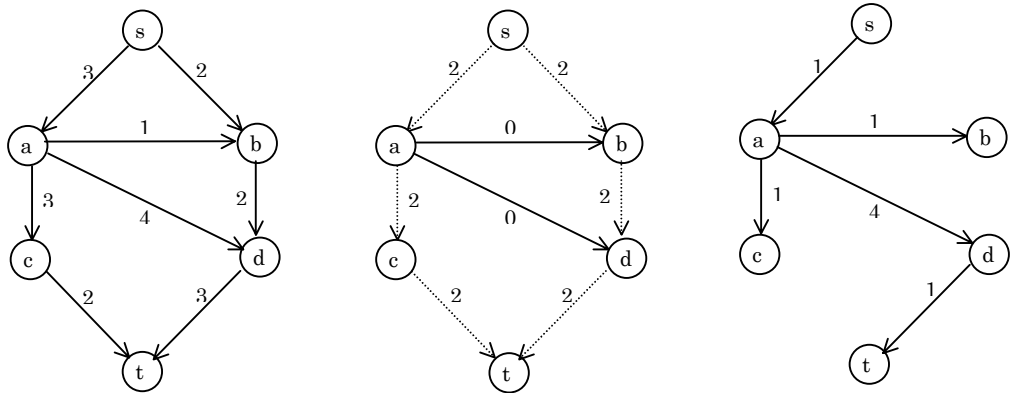



그림 9-23. 두개의 흐름단위가 *s*, *a*, *c*, *t* 에 따라 추가된후의 *G*, G_f, *Gr*

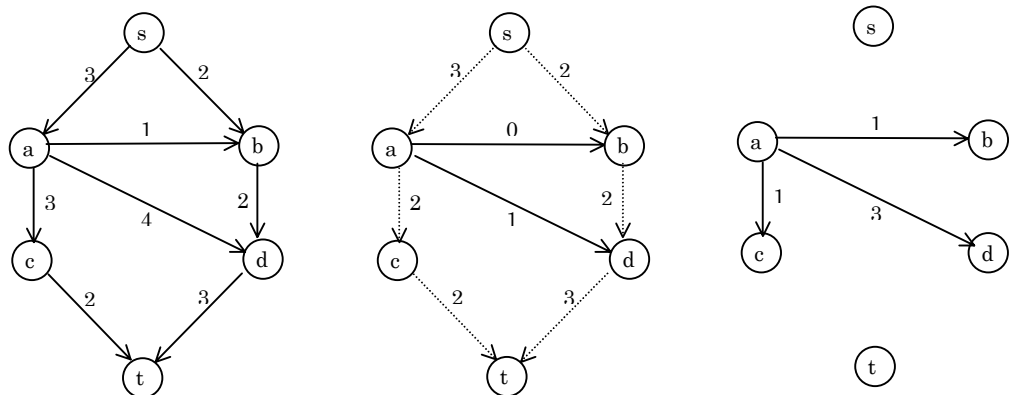



그림 9-24. 하나의 흐름단위가 *s*, *a*, *d*, *t* 에 따라 추가된후의 *G*, G_f, *Gr*(알고리듬끝)

이 경로는 세개의 흐름단위를 허용하며 따라서 좋은 선택안인것처럼 보인다. 그러나 이 선택안의 결과는 나머지그라프에서 s로부터 t에로 임의의 경로가 더이상 없으며 알고리듬은 최적인 풀이를 찾지 못했다는것을 보여 준다. 이것은 처리되지 않는 탐욕알고리듬의 한가지 실례이다. 그림 9-25는 알고리듬이 실패하는 리유를 보여 준다.

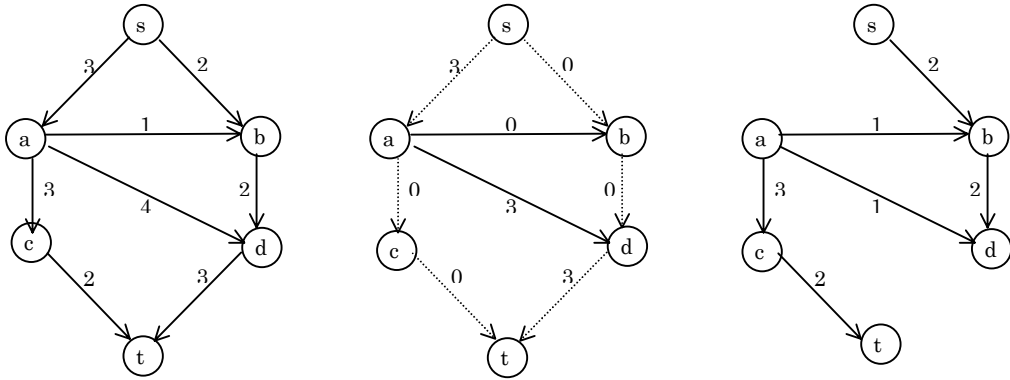



그림 9-25. 초기동작이 s, a, d, t 에 따라 세개의 흐름단위를 추가할 때의 G, G_f, Gr
(알고리듬은 정확한 처리를 하지 못하고 끝난다.)

이 알고리듬이 원만히 수행되도록 하기 위하여 그것을 변화시켜야 한다. 이를 위하여 흐름그라프에서 흐름 $f_{v,w}$를 가지는 모든 변 (v,w)에 대하여 용량 $f_{v,w}$를 가지는 나머지그라프(w,v)에 어떠한 변을 추가한다.

사실상 알고리듬은 흐름을 반대방향에로 다시 보내므로 처리를 원상태로 돌려 보낸다. 이것을 실례에서 보여 주었다. 초기그라프에서 시작하고 증가경로 s, a, d, t를 선택하여 그림 9-26과 같은 그라프를 얻는다.

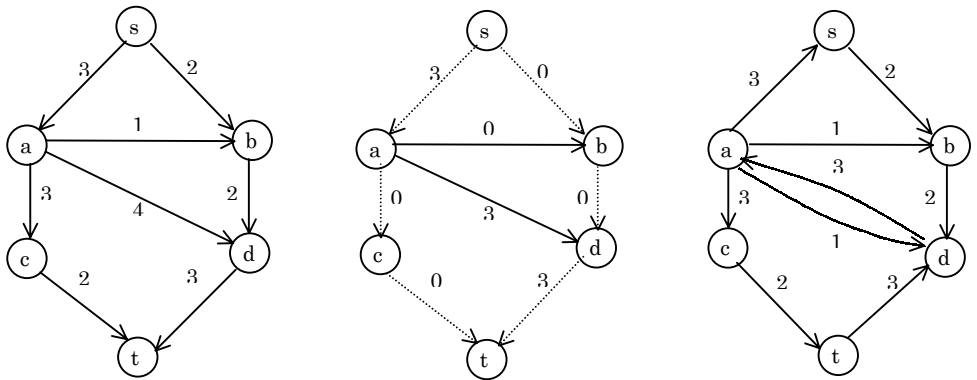



그림 9-26. 수정된 알고리듬을 리용하여 s, a, d, t에 따라
세개의 흐름단위가 첨부된후의 그라프들

흐름단위가 *a*로부터 *d*에로 통과하거나 3개이상의 흐름단위들이 반대로 통과할수 있다. 현재의 그 알고리듬은 두개의 흐름단위를 가지는 증가경로 *s*, *b*, *d*, *a*, *c*, *t*를 찾을수 있다.

*d*로부터 *a*에로 두개의 흐름단위가 통과함으로써 알고리듬은 변 (*a*, *d*)와 떨어 져 두개의 흐름단위를 가지며 본질적인 변화를 가져 온다. 그림 9-27은 새로운 그라프를 보여준다. 이 그라프에 증가경로가 없으므로 알고리듬은 끝난다. 변의 능력이 유리수라면 이 알고리듬은 항상 최대흐름으로 끝난다는것을 보여 준다. 이 증명은 어려우므로 이 책에서 취급하지 않는다.

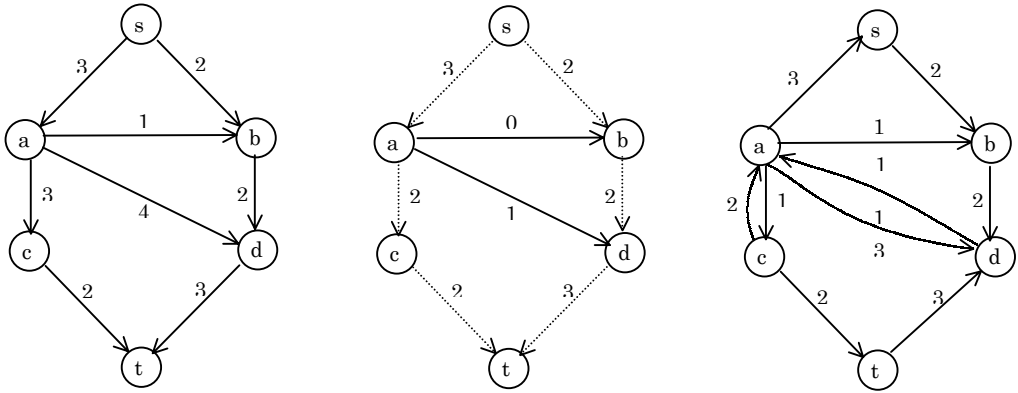



그림 9-27. 수정된 알고리듬을 리용하여 *s*, *t*, *d*, *a*, *c*, *t*에 따라 두개의 흐름단위가 첨부된후의 그라프들

실례가 비록 비순환으로 되여 있다고 해도 이것은 알고리듬이 동작하기 위한 요구로는 되지 않는다. 단지 간단히 하기 위하여 비순환그라프를 리용하였을뿐이다.

변의 능력이 모두 옹근수이고 최대흐름이 *f*라면 매 증가경로는 적어도 하나씩 흐름값이 증가하므로 *f*단계로서 충분히 수행된다. 그리고 무게없는최단경로알고리듬에 의하여 증가경로를 $O(|E|)$시간내에 찾을수 있으므로 전체 실행시간은 $O(f|E|)$이다.

이것이 좋은 실행시간으로 되지 못한다는 근거를 주는 전형적인 실례를 그림 9-28의 그라프에서 보여 준다. 매 변으로 1,000,000을 보낼수 있기때문에 2,000,000이 되는가 감시함으로써 최대흐름을 알수 있다. 우연적인 증가들은 *a*와 *b*를 련결하는 변을 포함하는 경로를 따라 계속 증가한다. 이것이 반복적으로 발생한다면 2.000,000증가가 요구되는데 그것은 오직 2로만 통과할 때이다.

이러한 문제점을 극복할수 있는 단순한 방법은 항상 흐름에서 가장 큰 증가를 허용하는 증가경로를 선택하는것이다. 이러한 경로를 찾는것은 무게붙은최단경로문제를 푸는것과 류사하며 딕스트라알고리듬에서 한개 행을 수정하여 이 목적을 달성할수 있다. Cap_{max}가 변의 최대능력이라면 $O(|E|logCap_{max})$증가가 최대흐름을 찾는데는 충분하다는것

을 보여 줄수 있다. 이 경우에 $O(|E|log|V|)$시간이 어떠한 증가경로에 대한 계산에 리용되므로 전체 한계는 $O(|E|^2log|V|log\ Cap_{max})$가 얻어 진다. 통과능력이 모두 작은 옹근수라면 $O(|E|^2log|V|)$로 줄어 든다.

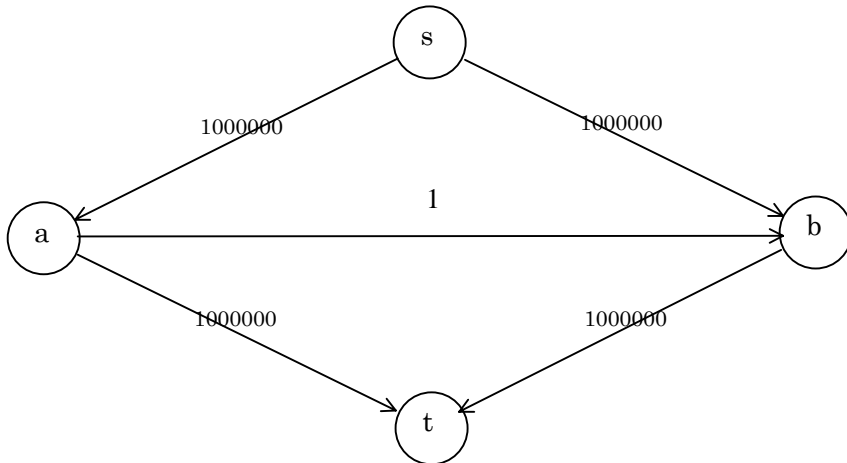



그림 9-28. 증가를 위한 전형적인 오유경우

증가경로를 선택하는 또 다른 방법은 항상 가장 적은 수의 변들을 가지는 경로를 취하는것이다. 이것은 이러한 수법으로 경로를 선택함으로써 작고 흐름제한을 가진 변이 경로상에 나타날 가능성이 적다는 기대를 가지게 한다. 이 규칙을 리용하여 $O(|E||V|)$증가단계들이 요구된다는것을 나타낼수 있다. 매 단계는 $O(|E|)$시간 걸리며 다시 무게없는최단경로알고리듬을 리용하여 실행할 때에 $O(|E|^2|V|)$한계를 가진다.

앞으로 자료구조를 더 개선하면 이 알고리듬의 수행에 실천적인 가능성을 주게 되며 거기에는 더 복잡한 여러가지 알고리듬들이 있게 된다. 오랜기간 개선되여 온 알고리듬의 시간한계들은 이 문제에 대한 현재의 한계보다 더 낮은 값을 가진다. $O(|E||V|)$알고리듬이 아직 없다고 해도 $O(|E||V|log(|V|^2/|E|))$과 $O(|E||V|+|V|)$한계를 가진 알고리듬들은 이미 제안되였다. 또한 특수한 경우 매우 좋은 한계를 주는 알고리듬도 있다. 실례로 $O(|E||V|^{1/2})$시간내에 그라프에서 최대흐름을 찾는 알고리듬이 있는데 이것은 원천지와 목적지를 제외한 모든 정점들이 능력이 1인 하나의 입력변이나 능력이 1인 하나의 출력변을 가진다는 속성을 가진다. 이 그라프들은 많은 응용프로그람에서 리용된다.

이러한 한계를 나타내는데 요구되는 분석은 좀 복잡하며 최악의 경우에 그 결과들이 실제로 실행시간에 어떻게 관계되는가하는것은 명백하지 않다. 실제적인 실행시간에 관계되면서도 좀 더 복잡한 문제는 **최소비용흐름**(*min-cost-flow*)문제이다. 매 변은 능력뿐아니라 단위흐름당 무게도 가진다. 이 문제는 모든 최대흐름중에서 하나의 최소무게흐름을 찾는것이다. 이 문제들은 둘다 연구되고 있다

제5절. 최소생성나무

이 절에서 고찰하는 다음 문제는 무방향그라프에서 **최소생성나무**(*minimum spanning tree*)를 찾는것이다. 이 문제는 방향그라프에서도 의미를 가지지만 확실히 더 어렵다. 보통 무방향그라프 G의 최소생성나무는 전체 값이 최소가 되도록 G의 모든 정점들을 련결하는 그라프변들로 이루어 진 나무이다. 최소생성나무는 G가 련결되여 있기만 하면 존재한다. 불충분한 알고리듬은 G가 련결되여 있지 않는 경우라고 해도 여기에서는 G가 련결되여 있다고 가정하고 련습문제로 남겨 놓는다.

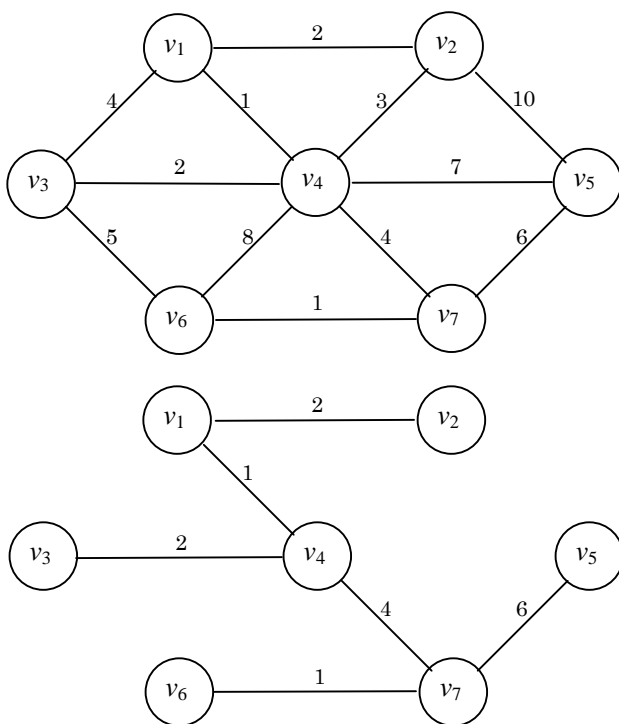



그림 9-29. 그라프 G와 그의 최소생성나무

그림 9-29의 두번째 그라프는 처음 최소생성나무이다(그것은 드문 일이지만 이것은 례외적이다.). 최소생성나무에서 변들의 개수는 $|V|$-1이다. 최소생성나무는 그것이 비순환이기때문에 하나의 나무이며 그리고 모든 정점을 망라하기때문에 생성한다고 하며 명백한 의미에서 최소로 된다. 최소한의 케블선으로 집에 전기선을 늘일 필요가 있다면 이것을 최소생성나무문제로 해결할수 있다. 임의의 생성나무 T에 대하여 T에 없는 변 e가 첨부된다면 순환이 이루어 진다. 순환그라프에서 임의의 변을 제거하면 이동은 생성나무의 속성을 되살린다. 생성나무의 무게는 e가 제거된 변보다 더 작은 값을 가진다면 작아 진

다. 생성나무가 만들어 질 때 첨부되는 변이 순환을 이루지 않는 최소무게중의 하나이라면 결과적인 생성나무의 무게를 개선할수 없다. 그것은 임의의 치환되는 변이 적어도 이미 생성나무에 있는 어떤 변만한 무게를 가지기때문이다. 이것은 최소생성나무문제에서 탐욕법이 리용된다는것을 보여 준다. 그 2개의 알고리듬은 최소변을 선택하는 방법이 다르다.

1. 프림알고리듬

최소생성나무를 만드는 한가지 방법은 련속적인 단계로서 나무를 증가시키는것이다. 매 단계에서 뿌리로서 한개 매듭을 골라 잡고 그와 관련된 변과 그에 따르는 정점을 나무에 첨부한다.

알고리듬의 어떤 시점에서 이미 나무에 포함된 정점들의 모임을 가지게 되는데 그 나머지정점들은 여기에 포함되지 않는다. 그때 알고리듬은 매 단계에서 변 (u,v)를 선택함으로써 나무에 첨부할 새로운 정점을 찾는데 (u,v)의 무게는 u가 생성나무에 있고 v는 생성나무에 없는 모든 변들중에서 가장 작은 값을 가진다. 그림 9-30은 알고리듬이 v_1에서 시작하여 최소생성나무를 어떻게 구축하는가를 보여 준다. 초기에 v_1은 변을 가지지 않는 나무의 뿌리이다. 매 단계에서 나무에 한개의 정점과 한개의 변을 첨부한다.

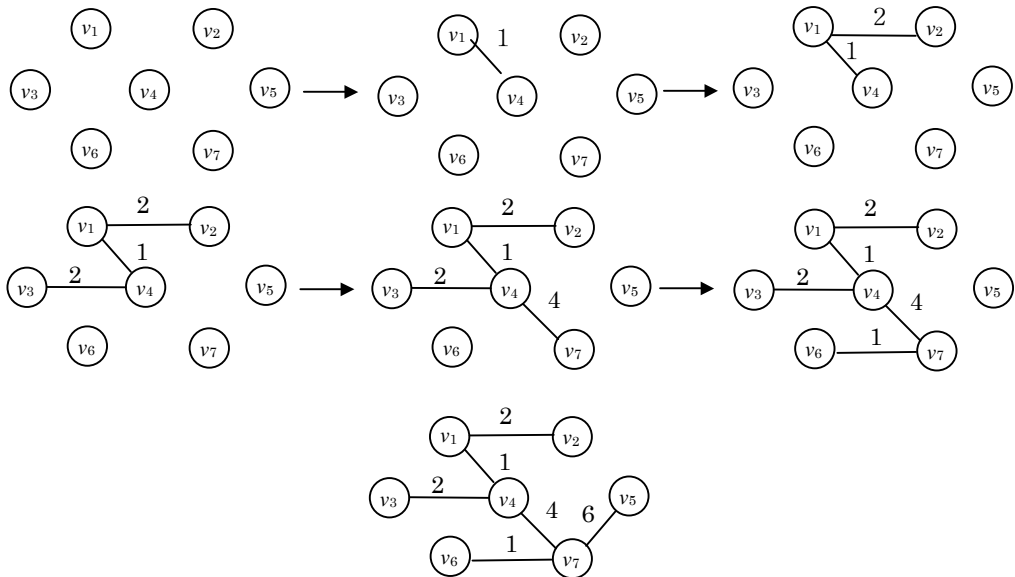



그림 9-30. 매 단계에서의 프림알고리듬

프림(Prim)알고리듬은 최단경로에서 딕스트라알고리듬과 본질적으로 같다. 앞에서

와 같이 매 정점에 대하여 값 d_v와 p_v 그리고 *known*인가 *unknown*인가에 대한 표식을 보관한다. d_v는 v를 *known*정점에 련결하는 가장 작은 변의 무게이며 p_v는 앞에서와 같이 d_v 것을 변화시키는 최종정점이다. 알고리듬의 나머지부분도 꼭 같다(d_v의 정의가 다르므로 갱신규칙을 제외하고). 이 문제에 대한 갱신규칙은 이전보다 더 쉽다. 정점 v가 선택된 다음 v와 린접인 매 *unknown* w에 대하여 $d_w=min(d_w,c_{w,u})$이다.

표 9-11. 프림알고리듬에 리용되는 표의 초기구성

v	*Known*	d_v	p_v
v_1	F	0	0
v_2	F	∞	0
v_3	F	∞	0
v_4	F	∞	0
v_5	F	∞	0
v_6	F	∞	0
v_7	F	∞	0

표의 초기구성을 표 9-11에 보여 주었다. v_1이 선택되고 v_2, v_3, v_4가 갱신된다. 이로부터 얻어 지는 표를 표 9-12에서 보여 주었다. 선택된 다음 정점은 v_4이다. 모든 정점은 v_4와 린접이다. v_1은 *known*이기때문에 조사되지 않는다.

v_2도 변화되지 않는데 그것은 d_v=2이고 v_4로부터 v_2에로의 변무게가 3이기때문이다. 나머지는 모두 갱신된다. 표 9-13은 결과표를 보여 준다. 선택된 다음 정점은 v_2이다.

.

표 9-12. v_1이 *known*선언된후의 표

v	*Known*	d_v	p_v
v_1	T	0	0
v_2	F	2	v_1
v_3	F	4	v_1
v_4	F	1	v_1
v_5	F	∞	0
v_6	F	∞	0
v_7	F	∞	0

표 9-13. v_4가 *known*선언된후의 표

v	*Known*	d_v	p_v
v_1	T	0	0
v_2	F	2	v_1
v_3	F	2	v_4
v_4	T	1	v_1
v_5	F	7	v_4
v_6	F	8	v_4
v_7	F	4	v_4

표 9-14. v_2,v_3이 *known*선언된후의 표

v	*Known*	d_v	p_v
v_1	T	0	0
v_2	T	2	v_1
v_3	T	2	v_4
v_4	T	1	v_1
v_5	F	7	v_4
v_6	F	5	v_3
v_7	F	4	v_4

표 9-15. v_7이 *known*선언된후의 표

v	*Known*	d_v	p_v
v_1	T	0	0
v_2	T	2	v_1
v_3	T	2	v_4
v_4	T	1	v_1
v_5	F	6	v_7
v_6	F	1	v_7
v_7	T	4	v_4

이것은 어떤 거리들에 영향을 주지 않는다. 그다음 v_3이 선택되는데 이것은 표 9-14에서 보여 주는것처럼 v_6의 거리에 영향을 준다. 표 9-15는 v_7의 선택에 귀착되며 v_6과 v_5가 조정되도록 한다. v_6과 그다음 v_5가 선택되면 알고리듬을 완료된다.

최종표는 표 9-16에 보여 주었다. 생성나무에 있는 변들은 표로부터 읽어 낼수 있다. 즉 $(v_2, v_1), (v_3, v_4), (v_4, v_1), (v_5, v_7), (v_6, v_7), (v_7, v_4)$가 얻어 진 그 전체 무게값은 16이다

표 9-16. v_6과 v_5가 선택된후의 표

v	*Known*	d_v	p_v
v_1	T	0	0
v_2	T	2	v_1
v_3	T	2	v_4
v_4	T	1	v_1
v_5	T	6	v_7
v_6	T	1	v_7
v_7	T	4	v_4

이 알고리듬의 총체적인 실현방법은 가상적으로 딕스트라알고리듬과 동일하며 딕스트라알고리듬의 분석에 대해서 설명된 모든것이 여기에서도 적용된다. 프림알고리듬이 무방향그라프에서 실행된다는것을 알고 따라서 그것을 코드화할 때 모든 변을 2개의 린접표에 넣는다. 더미를 리용하지 않을 때 그 실행시간은 $O(|V|^2)$으로서 이것은 조밀한 그라프에 대하여 최적으로 되며 2진더미를 리용하면 그 실행시간은 $O(|E|log|V|)$으로서 성긴그라프에서 효률적이다.

2. 크루스칼알고리듬

두번째 탐욕알고리듬은 가장 작은 무게순서로 변들을 계속 선택해 나가면서 그것들이 만일 순환을 일으키지 않는다면 선택된 변을 생성나무정점들의 모임에 넣는것이다. 앞의 실례와 같은 그라프상에서 알고리듬처리과정은 표 9-17에서 보여 준것과 같다.

표 9-17. G에 대한 크루스칼알고리듬의 동작

매듭	무게	작용
(v_1, v_4)	1	접수
(v_6, v_7)	1	접수
(v_1, v_2)	2	접수
$(v_3, v_4$	3	거절
(v_1, v_3)	4	거절
(v_4, v_7)	4	접수
(v_3, v_6)	5	거절
(v_5, v_7)	6	접수

형식적으로 **크루스칼**(*kruskal*)알고리듬은 나무의 집합 즉 수림을 유지한다. 초기에 $|V|$개의 단일매듭나무들이 있다. 하나의 변을 첨부하여 2개의 나무를 하나로 병합한다. 알고리듬이 완료되면 오직 하나의 나무만 남게 되는데 이것이 최소생성나무이다. 그림 9-31은 변들이 수림에 첨부되는 순서를 보여 준다.

변들이 충분히 접수되면 알고리듬이 완료된다. 이 알고리듬은 변 (u,v)를 받아 들여야 하는가 혹은 거절해야 하는가를 쉽게 결정하게 한다. 적당한 자료구조는 앞의 장에서

고찰한 union/find알고리듬이다.

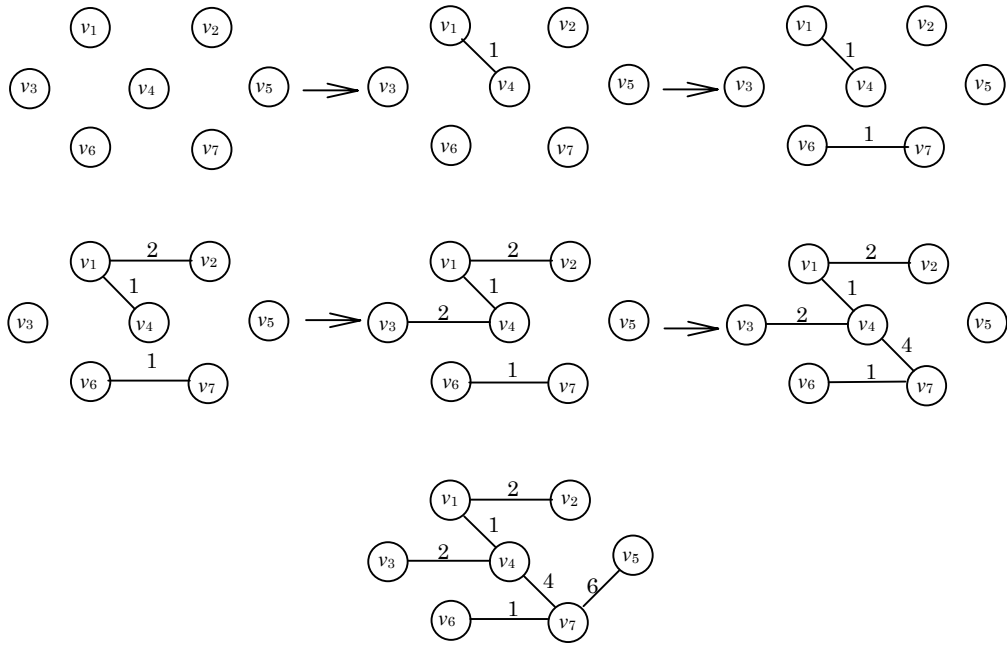



그림 9-31. 크루스칼알고리듬의 매 단계

여기에서 우리가 리용하여야 할 한가지 사실이 있다. 그것은 처리과정의 임의의 시점에서 두개의 정점들이 현재 생성나무수림에 련결되여 있기만 하면 같은 모임에 속한다는것이다. 따라서 매 정점은 초기에 그 자체모임에 들어 있다. u와 v가 같은 모임에 있다면 그 변은 접수되지 않는다. 그 리유는 그것들이 이미 련결되여 있으므로 (u, v)를 첨부하면 순환을 형성하기때문이다. 만일 련결되여 있지 않으면 그 변이 접수되고 그리고 union은 u와 v를 포함하는 두개의 모임에 대하여 처리된다. 이것은 모임이 변하지 않도록 유지한다고 보는것이 좋다. 그것은 변 (u, v)가 생성수림에 첨부되자마자 w가 u에, x가 v에 련결된다면 x와 w는 곧 련결되여야 하며 따라서 그것들은 같은 모임에 속하기때문이다.

그 변들은 선택에 편리하도록 정렬되여야 하지만 선형시간내에 어떤 더미구조를 구축하도록 하는것이 훨씬 더 좋은 방법이다. 그때 deleteMin연산들은 변들을 순서대로 검사하도록 한다. 모든 변들이 항상 처리될수 있다고 해도 일부 변들만이 그 알고리듬을 끝내기전에 검사되여야 한다. 실례로 특별한 정점 v_8과 무게가 100인 변 (v_5, v_8)이 있다면 모든 변들이 검사되여야 한다. 프로그람 9-10에 있는 kruskal함수는 최소생성나무를 탐색한다.

이 알고리듬의 최악의 경우 실행시간은 $O(|E|log|E|)$인데 이것은 더미연산들에 의하여 결정된다. $|E|=O(|V|^2)$이므로 이 실행시간은 실제로 $O(|E|log|V|)$이다. 실천적으로 이 알고리듬은 지적된 시간한계보다 훨씬 더 빠르다.

```
        void Graph::kruska1( )
        {
            int edgesAccepted;
            DisjSet s( NUM_VERTICES );
            PriorityQueue h( NUM_EDGES );
            Vertex u, v;
            SetType uset, vset;
            Edge e;

/*1*/           h = readGraphIntoHeapArray( );
/*2*/           h.buildHeap( );

/*3*/           edgesAccepted = 0;
/*4*/           whi1e( edgesAccepted < NUM_VERTICES - 1 )
                {
/*5*/               h.de1eteMin( e ); // Edge e = (u, v)
/*6*/               uset = s.find( u );
/*7*/               vset = s.find( v );
/*8*/               if( uset != vset )
                    {
                        // Accept the edge
/*9*/                   edgesAccepted++;
/*10*/                  s.unionSets( uset, vset );
                    }
                }
        }
```

프로그람 9-10. 크루스칼알고리듬의 가상코드

제6절. 깊이우선탐색의 응용

깊이우선탐색은 선뿌리순회의 일반적인 경우이다. 어떤 정점 v에서 시작하여 v를 처리하고 그다음 v와 린접인 모든 정점들을 재귀적으로 순회한다. 이런 처리가 어떤 나무에서 실행된다면 $|E|=\Theta(|V|)$이므로 나무의 모든 정점들은 총체적으로 $O(|E|)$시간에 체계적으로 방문된다. 만일 임의의 그라프에서 이러한 처리가 실행된다면 순환을 피하도록 주의하여야 한다. 이를 위하여 정점 v를 방문하면 그것이 방문되였다는 **표식기호**(*mark*)를 붙인다. 그리고 이런 표식기호가 붙지 않은 모든 린접점들에 대하여서는 재귀적으로 깊이우선탐색을 진행한다. 암시적으로 무방향그라프에서 모든 정점 (v,w)는 린접표에 두번

즉 한번은 (v, w)로서 다른 한번은 (w, v)로서 나타난다고 하자. 프로그람 9-11의 루틴은 깊이우선탐색을 실행하는것으로서 일반적인 표기법으로 서술한 하나의 형판이다.

```
void Graph::dfs( Vertex v )
{
        v.visited = true;
        for each w adjacent to v
            if( !w.visited )
                dfs( w );
}
```

프로그람 9-11. 깊이우선탐색에 관한 형판

매 정점에 대하여 visited자료성원은 false로 초기화된다. 방문하지 않은 매듭들에 대하여서만 수속을 재귀적으로 호출함으로써 무한순환에 빠지지 않도록 한다. 그라프가 무방향성이면서 련결되지 않았거나 방향성이면서 강하게 련결되지 않았다면 이 방법으로는 일부 정점들을 방문하는데서 실패할수 있다. 그다음 표식기호가 붙지 않은 매듭에 대하여 탐색하고 거기에 깊이우선순회를 적용한다. 그리고 표식기호가 붙지 않은 매듭이 없을 때까지 이 과정을 계속한다. 이 방법은 매 변이 한번만 처리되기때문에 그 순회시간은 린접표를 리용할 때 $O(|E|+|V|)$으로 된다.

1. 무방향그라프

그 임의의 매듭으로부터 시작하여 어떤 무방향그라프가 련결되였다면 깊이우선탐색은 임의의 매듭에서 시작하여 그라프의 모든 매듭을 방문할수 있다. 깊이우선탐색이 모든 매듭을 방문하기만 하면 무방향그라프는 련결된다. 이 검사는 적용하기 쉬우므로 여기서 고찰하는 그라프들은 련결된것으로 가정한다. 만일 그라프가 련결되지 않았다면 련결된 성분들을 모두 찾고 매개 련결성분들에 대하여 차례로 알고리듬을 적용할수 있다.

실례로서 그림 9-32의 그라프의 정점 A에서 깊이우선탐색을 시작한다고 하자. A를 방문된것으로 표시하고 dfs(B)를 재귀적으로 호출한다. dfs(B)는 B를 방문한것으로 표시하며 dfs(C)를 재귀적으로 호출한다. dfs(C)는 C를 방문한것으로 표시하고 재귀적으로 dfs(D)를 호출한다. dfs(D)는 A와 B 둘다 방문할수 있지만 둘다 방문한것으로 표시되여 있으므로 재귀호출이 더는 일어 나지 않는다. dfs(D)는 또한 C가 린접이지만 표식되여 있다는것을 알게 되므로 거기에서 재귀호출이 더는 일어 나지 않으며 dfs(D)는 dfs(C)에로 다시 되돌아 온다. dfs(C)는 B와 린접이라는것을 알고 그것을 무시하고 이미 알려 지지 않은 린접점 E를 찾게 되므로 dfs(E)를 호출한다. dfs(E)는 E를 방문한것으로 표시하고 A와 C를 무시하며 그리고 dfs(C)에로 되돌아 온다. dfs(C)는 dfs(B)에로 돌아 온다. dfs(B)는 A와 D 둘다를 무시하고 되돌아 온다. dfs(A)는 D와 E 둘다를 무시하고 돌아 온다(실

제로 매 변을 두번 거친다. 즉 (v, w)로서 한번 거치고 다시 (w, v)로서 한번 거친다. 그러나 이것은 매개 린접목록입력점당 실제로는 한번으로 된다.).

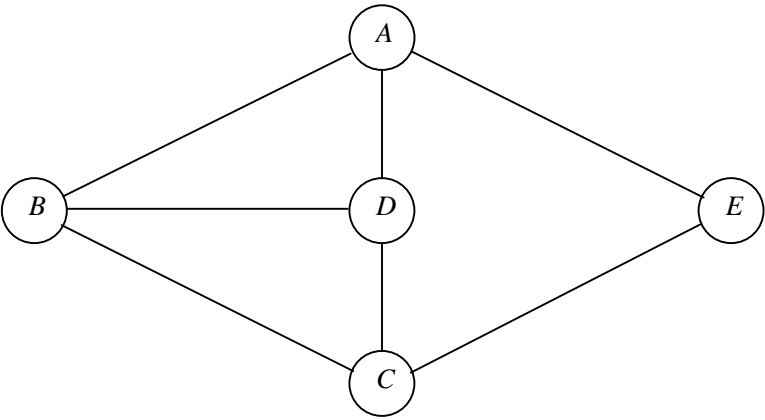



그림 9-32. 무방향그라프

깊이우선생성나무(*depth-first spanning tree*)로서 이 단계들을 도형적으로 설명하자. 나무의 뿌리는 처음으로 방문된 정점 A이다. 그라프에서 매 변 (v, w)는 그 나무에 있다. 만일 (v, w)를 처리할 때 w가 표시되지 않았다는것을 알게 되거나 (w, v)를 처리할 때 v가 표시되지 않았다는것을 알게 되면 이것을 나무의 변으로 지적한다. 만일 (v, w)를 처리할 때 w가 이미 표식이 붙었다는것을 알게 되고 (w, v)를 처리할 때 v가 이미 표식이 붙었다는것을 알게 되면 이 《변》이 실제로 나무의 부분이 아니라는것을 나타내기 위하여 **뒤변**(*back edge*)이라고 하는 점선으로 표시한다. 그림 9-32에 있는 그라프에 대한 깊이우선탐색을 그림 9-33에서 보여 주었다.

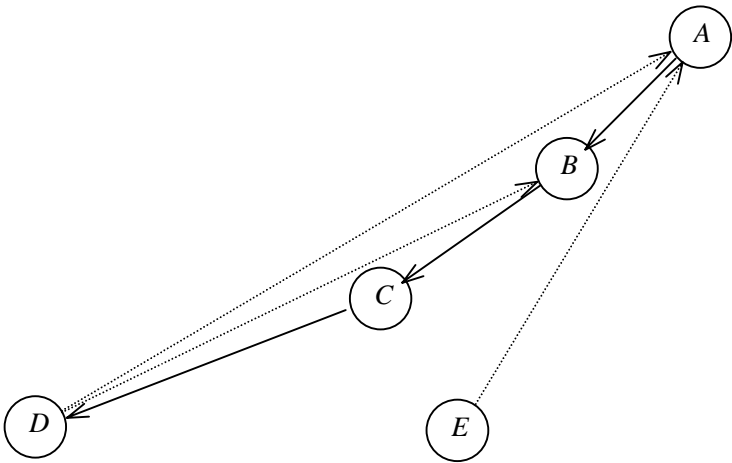



그림 9-33. 그림 9-32 의 그라프에서 깊이우선탐색

이 나무에 대한 순회과정을 모의하자. 나무의 변들만 리용하면 나무의 선뿌리순회순서번호는 정점에 표식기호를 붙인 순서를 나타낸다. 그라프가 련결되지 않았다면 모든 매듭(그리고 변)들에 대한 처리는 dfs를 여러번 호출하며 매번 호출할 때마다 하나의 나무를 만든다. 이 전체적인 모임을 **깊이우선생성수림**(*depth-first spannig forest*)이라고 한다.

2. 쌍련결성

만일 련결된 무방향그라프에서 정점들의 삭제가 그라프의 나머지를 분리하지 않는다면 그 그라프는 쌍련결되였다고 한다. 우의 실례에서 그라프는 쌍련결되여 있다. 만일 매듭들이 콤퓨터들이고 변들이 련결선이라고 하자. 그때 어떤 말단콤퓨터가 있다면 콤퓨터망전자우편은 말단콤퓨터를 제외하고 영향을 받지 않는다. 이와 류사하게 수화물운반체계가 쌍련결이면 어떤 말단에서 서로 엇갈리는 길을 가지는 사용자들은 항상 혼란된다.

그라프가 쌍련결되지 않았을 때 정점들의 삭제로 그라프를 분리시킬 때 바로 그 정점을 **분리점**(*articulation points*)이라고 한다. 이 매듭들은 많은 응용에서 중요하다. 그림 9-34의 그라프는 쌍련결되여 있지 않다. C와 D는 분리점이다. C의 제거는 G를 분리시키며 D의 제거는 그라프에서 E와 F를 분리시킨다.

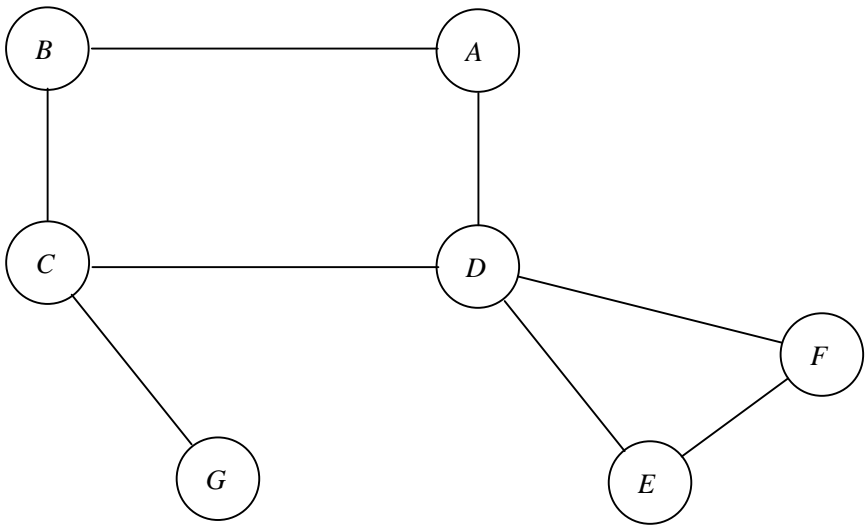



그림 9-34. 분리점 C와 D를 가진 그라프

깊이우선탐색은 련결그라프에서 모든 분리점을 찾기 위한 선형시간알고리듬을 준다. 먼저 임의의 정점에서 출발하여 깊이우선탐색을 진행하여 매듭들이 방문될 때 그 매듭들에 번호를 붙인다. 매 정점 v에 대하여 선뿌리순회순서번호 $Num(v)$를 호출한다.

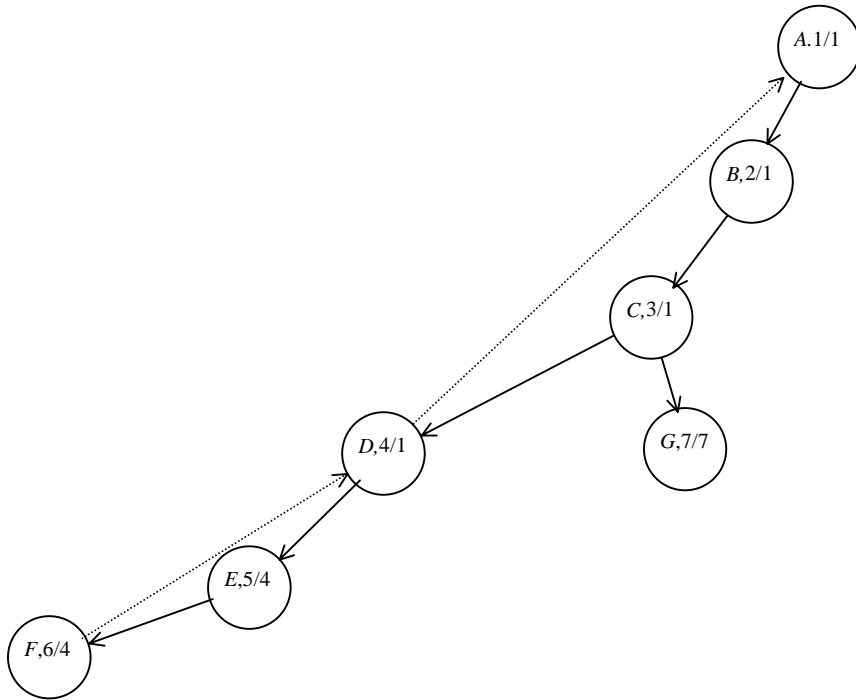



그림 9-35. *Num*과 *Low*를 가진 우에서의 그라프에 대한 깊이우선나무

그다음 깊이우선탐색생성나무에 있는 모든 정점 *v*에 대하여 가장 작은 번호를 가진 정점을 계산하는데 이 처리를 *Low*(*v*)라고 한다. 이것은 0 또는 그이상의 나무변들을 가짐으로써 *v*로부터 도달할수 있으며 그 순서로 하나의 뒤변을 설정할수 있다. 그림 9-35에서 깊이우선탐색나무는 먼저 선뿌리순회번호를 보여 주고 그다음 우에 서술된 규칙에 따라 이룰수 있는 가장 작은 번호를 가진 정점을 보여 준다.

A, *B*, *C*에 의해서 도달할수 있는 가장 작은 번호를 가진 정점은 정점 1(A)인데 그것은 *D*에로의 나무변들과 *A*에 대한 하나의 뒤변을 다 취할수 있기때문이다. 우리는 깊이우선생성나무에 대한 후뿌리순회를 수행함으로써 *Low*를 효률적으로 계산할수 있다. *Low* 정의에 의하면 *Low*(*v*)는 다음과 같은것의 최소값이다.

① *Num*(*v*)
② 모든 뒤변 (*v*, *w*)들중에서 가장 작은 *Num*(*w*)
③ 모든 나무변 (*v*, *w*)들중에서 가장 작은 *Low*(*v*)

첫 조건은 그 어떤 변도 취하지 않는것이다. 두번째 방법은 나무의 변들은 하나도 선택하지 않고 뒤변만 선택하는것이며 세번째 방법은 어떤 나무변들을 선택하고 가능하면 하나의 뒤변을 선택하는것이다. 이 세번째 방법은 재귀호출로 간단히 서술된다. *Low*(*v*)를 계산하기전에 *v*의 모든 자식들에 관한 *Low*를 평가하여야 하는데 이것은 후뿌리

순회로 처리할수 있다. 임의의 변 (v, w)에 대하여 $Num(v)$와 $Num(w)$를 검사하여 그것이 나무변인가 혹은 순수 뒤변인가를 알수 있다. 따라서 $Low(v)$를 계산하기 쉽다. 여기에서는 단순히 v의 린접목록을 주사하여 내려 가면서 적당한 규칙을 리용하여 최소값을 기록하여 둔다. 이 모든것을 계산하는데 $O(|E|+|V|)$시간이 걸린다.

모든 처리는 분리점들을 찾기 위하여 이 정보를 리용하여야 한다. 뿌리가 하나이상의 자식을 가지고 있기만 하면 뿌리는 분리점이다. 그것은 뿌리가 2개의 자식을 가지면 그 뿌리의 삭제는 다른 부분나무들에 있는 매듭들을 분리하며 뿌리가 하나의 자식만을 가지고 있다면 뿌리의 삭제는 단순히 그 뿌리를 분리하기때문이다. 임의의 다른 정점 v는 v가 $Low(w) \geqq Num(D)$인 어떤 자식 w를 가지기만 하면 분리점으로 된다. 특정한 검사에 필요하기때문에 이 조건이 항상 뿌리에서 만족된다는것에 주의하시오.

증명의 if부분은 알고리듬이 결정하는 분리점들 다시말하여 C와 D를 조사하면 명백하다. D는 $Low(E) \geqq Num(D)$이고 둘다 4이므로 자식 E를 가진다. 따라서 E가 D우의 임의의 매듭에 이르려면 오직 한가지 방법밖에 없다. 즉 D를 통하여 가는것이다. 이와 마찬가지로 C는 분리점인데 그것은 $Low(G) \geqq Num(C)$이기때문이다. 이 알고리듬이 정확하다는 것을 증명하기 위하여 알고리듬의 if부분이 참이라는것을 보여 주어야 한다. 이것은 련습문제로 남겨 놓는다. 두번째 실례로 C에서 깊이우선탐색을 시작하여 같은 그라프상에서 이 알고리듬을 적용한 결과를 보여 준다(그림 9-36).

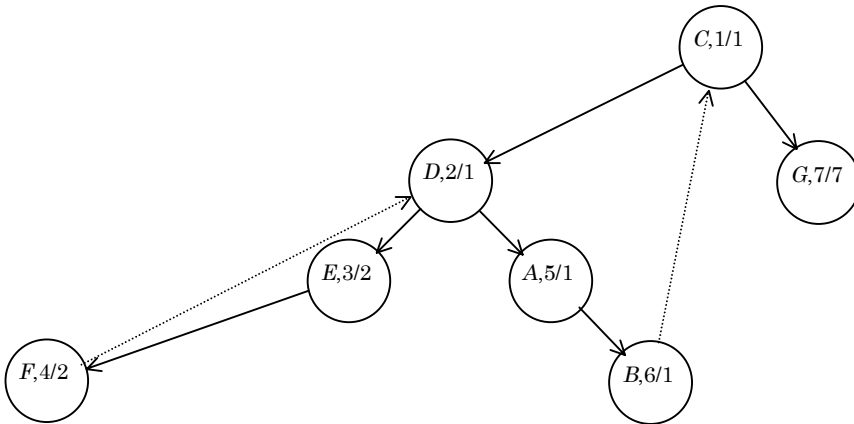



그림 9-36. C에서부터 깊이우선탐색을 진행하여 결과를 주는 깊이우선나무

이 알고리듬을 실현하기 위한 가상코드를 주는것으로써 이에 대한 설명을 끝낸다. Vertex클라스는 자료성원들 visited(false로 초기화된), num, low, parent를 포함한다. 또한 counter라고 부르는 클라스변수를 보존하는데 이것은 선뿌리순회번호들인 num을 할당하기 위하여 1로 초기화된다. 또한 뿌리에 대하여 쉽게 실현되는 검사는 생략한다.

이미 서술한바와 같이 이 알고리듬은 num을 계산하기 위하여 선뿌리순회를 수행하여 실현할수 있으며 Low를 계산하기 위해서는 후뿌리순회를 진행함으로써 실현할수 있

다. 세번째 순회는 어느 정점들이 분리점조건을 만족하는가를 검사하기 위하여 리용할수 있다. 그러나 세번째 순회를 진행하는것은 비효률적이다. 첫번째 단계를 프로그람 9-12에서 보여 주었다.

```
          /**..
           * Assign num and compute parents.
           */
           void Graph: :assignNum( Vertex v )
           {
                Vertex w;

/*1*/           v.num = counter++;
/*2*/           v.visited = true;
/*3*/           for each w adjacent to v
/*4*/               if( !w.visited )
                    {
/*5*/                     w,parent = v;
/*6*/                     assignNum( w );
                     }
           }
```

프로그람 9-12. 정점들에 *Num*을 할당하는 루틴

후뿌리순회들인 두번째와 세번째 단계들은 프로그람 9-13에서 보여 준 코드로서 실현할수 있다. 8행은 특수한 경우를 조종한다. *w*가 *v*의 린접이라면 *w*에로의 재귀호출은 *w*에 린접인 *v*를 찾는다. 이것은 이미 고찰된 유일한 변이 뒤변이 아니며 무시되여야 한다. 다른 한편 수속은 알고리듬에 지적된것처럼 low와 num입력점들의 최소값을 계산한다.

```
           /**
            *. Assign low; also check for articulation points.
            */
           void Graph::assignLow( Vertex v )
           {
                Vertex w;

/*1*/              v.low = v.num;         // Rule 1
/*2*/              for each w adjacent to v
                   {
/*3*/                   if( w.num > v.num )    // Forward edge
                        {
/*4*/                        assignLow( w );
/*5*/                        if( w.1ow >= v.num )
/*6*/                        cout << v << “is an articulation point”<< endl;
/*7*/                        v.low = min( v.low, w.1ow );    // Rule 3
```

```
                    }
                    else
/*8*/                       if( v. parent != w )    // Back edge
/*9*/                           v.tow = min( v.low, w.num );    // Rule 2
                }
```

프로그람 9-13. *Low*를 계산하고 분리점들을 검사하기 위한 가상코드

거기에는 순회가 선뿌리순회여야 하는지 후뿌리순회여야 하는지를 결정하는 규칙이 없다. 그것은 재귀호출을 수행하기전과 수행한 다음의 두가지 경우를 처리하여 결정할수 있다. 프로그람 9-14의 수속은 수속 findArt를 만들기 위하여 간단한 수법으로 2개의 루틴 assignNum과 assignLow를 결합한다.

```
            void Graph: :findArt( Vertex v )
            {
            Vertex w;

/*1*/         v.visited = true;
/*2*/         v.low = v.num = counter++;    // Rule 1
/*3*/         for each w adjacent to v
            {
/*4*/             if( !w.visited )    // Forward edge
                  {
/*5*/                 w.parent = v;
/*6*/                 findArt( w );
/*7*/                 if( w.1ow >= v.num )
/*8*/                         cout<< v << " is an articulation point"<< endl;
/*9*/                 v.low = min( v.low, w.1ow );    // Rule 3
                  }
                  else
/*10*/            if( v. parent != w )    // Back edge
/*11*/                v.low = min( v.low, w.num );    // Rule 2
              }
          }
```

프로그람 9-14. 하나의 깊이우선탐색에서 분리점들에 대한 검사(뿌리에 대한 검사는 제외)

3. 오일레르의 회로

그림 9-37에 있는 3개의 도형을 고찰하자. 이 문제는 펜으로 매 선을 정확히 한번 그어서 이 도형을 재구성하는것이다. 그리기가 진행되는 동안 펜은 종이에서 뗄수 없다.

매우 어려운 문제는 시작점에서 마지막펜을 떼는것이다. 이 수수께기는 놀랍게도 단순한 풀기법을 가진다. 그것을 풀어 보고 싶으면 한번 해보시오.

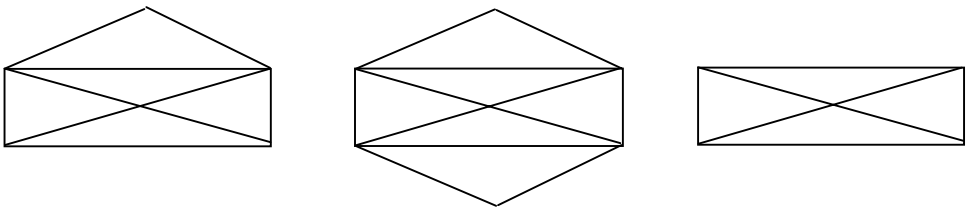



그림 9-37. 세개의 도형들

첫 도형은 시작점이 오른쪽 혹은 왼쪽 아래 모서리인 때에만 그려질수 있으며 시작점에서 끝내기는 불가능하다. 두번째 도형은 시작점과 같은 점에서 끝나도록 쉽게 그릴수 있다. 그러나 세번째 도형은 수수께끼의 파라메터로서는 전혀 풀수 없다.

매 교차점에 정점을 할당함으로써 이 문제를 그라프리론문제로 변환할수 있다. 그러면 자연적수법으로 그림 9-38과 같이 변들이 할당된다.

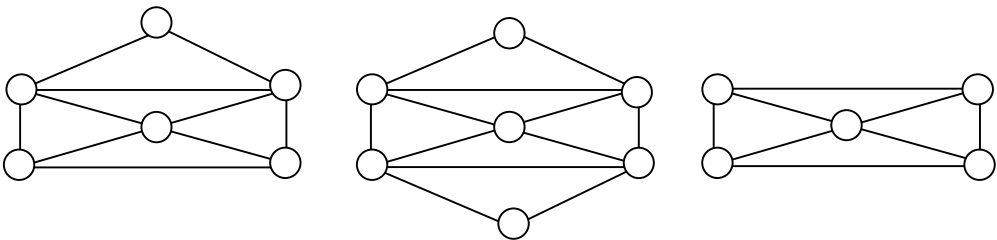



그림 9-38. 수수께끼를 그라프로 변환

이 변환을 진행한 다음 매 변을 정확히 한번만 방문하는 그라프경로를 찾아야 한다. 《극단한 난문제》를 해결하자면 모든 변을 정확히 한번만 방문하는 순환을 찾아야 한다. 이 그라프문제는 오일레르에 의하여 1736년에 해결되였다. 그리고 그라프리론의 시작으로 기록되였다. 따라서 이 문제를 일반적으로 구체적인 문제상태에 따라서 **오일레르경로**(*Euler path*) 때때로 **오일레르순회문제**(*Euler tour problem*) 혹은 **오일레르회로문제**(*Euler circuit problem*)라고 한다. 오일레르경로와 오일레르회로문제는 약간 다르지만 기본적으로 같은 풀기법을 가진다. 따라서 이 절에서 오일레르회로문제를 고찰한다.

이것을 해결하는 첫번째 방법은 오일레르회로가 시작정점에서 끝나야 하며 이것은 그라프가 련결되여 있고 매 정점이 짝수차원(변들의 수)을 가질 때에만 가능하다는것이다. 이것은 오일레르회로에서 임의의 정점이 입력되고 거기에서 나기때문이다. 임의의 정

점 v가 홀수차원을 가질 때 우연적으로 v에로의 하나의 변만이 방문하지 못하게 된다. 그리고 그것은 v에서 정지되게 된다.

정확히 2개의 정점이 홀수차원을 가질 때 모든 변을 방문해야 하지만 그 시작정점에로 돌아 갈 필요가 없는 오일레르순회는 홀수차원정점들중 하나에서 출발하여 다른 정점에서 끝나는 처리가 가능하다. 2이상의 정점이 짝수차원이라면 오일레르회로는 불가능하다. 이상의 고찰은 오일레르회로존재에 대한 필요한 조건을 준다. 그러나 이 속성을 만족하는 모든 련결그라프가 오일레르회로로 된다는것을 의미하지는 않으며 그것을 찾는 방법에 대한 지침을 준다. 또한 이 조건이 오일레르회로가 되기 위한 필요충분조건이라는 것을 의미한다. 즉 임의의 련결그라프에서 그의 모든 정점들이 짝수차원을 가진다면 오일레르회로로 된다. 또한 그 회로를 선형시간내에 찾을수 있다.

선형시간내에 필요충분조건을 검사할수 있으므로 오일레르회로가 존재한다는것을 알고 있다고 가정하자. 그러면 기본알고리듬은 깊이우선탐색을 진행하는것이다. 놀랍게도 처리되지 않는 많은 명백한 해답들이 있다. 이들중 일부는 련습문제에서 제기된다.

기본문제는 그라프의 한 부분을 방문하고 시작점으로 돌아 오는것이다. 시작점에서 나가는 모든 변들이 리용되였다면 그라프부분을 순회하지 못한다. 이것을 수정하는 가장 쉬운 방법은 이 경로우에서 순회되지 않은 변을 가지는 첫 정점을 찾고 다시한번 깊이우선탐색을 진행하는것이다. 이것은 또 다른 회로를 주며 본래문제로 합치여 진다. 이것은 모든 변들을 순회할 때까지 계속된다.

실례로 그림 9-39에 준 그라프를 고찰하자. 이 그라프가 오일레르회로를 가진다는것을 쉽게 알수 있다. 정점 5에서 출발하여 회로 5, 4, 10, 5를 순회한다고 가정하자. 그때 그라프의 대부분은 여전히 순회되지 않았다. 이러한 상황을 그림 9-40에서 보여 주었다.

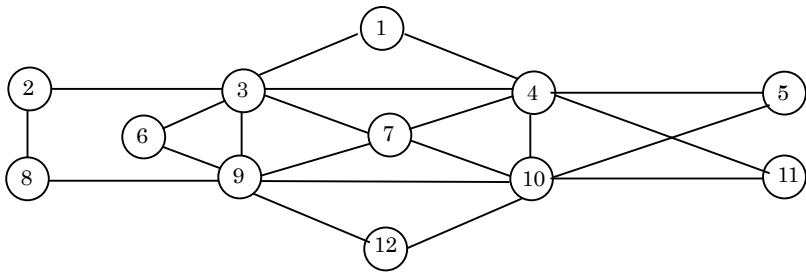



그림 9-39. 오일레르회로문제에 관한 그라프

그다음 정점 4에서 처리를 계속하는데 정점 4는 아직 탐색되지 않은 변들을 가지고 있다. 하나의 깊이우선탐색이 경로 4, 1, 3, 7, 4, 11, 10, 7, 9, 3, 4에 대하여 진행된다. 이 경로를 앞에서의 5, 4, 10, 5경로에 이어서 붙인다면 새로운 경로 5, 4, 1, 3, 7, 4, 11, 10, 7, 9, 3, 4, 10, 5를 얻는다.

이후에 남아 있는 그라프를 그림 9-41에서 보여 주었다. 이 그라프에 있는 모든 정점들은 짝수차원을 가져야 한다는데 주목하므로 첨부되는 순환을 찾는다. 나머지그라프는 련결되지 않을수 있지만 이것은 중요한것이 아니다. 순환되지 않은 변들을 가지는 경로상의 다음정점은 정점 3이다. 가능한 회로는 그때 3, 2, 8, 9, 6, 3이다. 련결되면 이것은 경로 5, 4, 1, 3, 2, 8, 9, 6, 3, 7, 4, 11, 10, 7, 9, 3, 4, 10, 5을 준다.

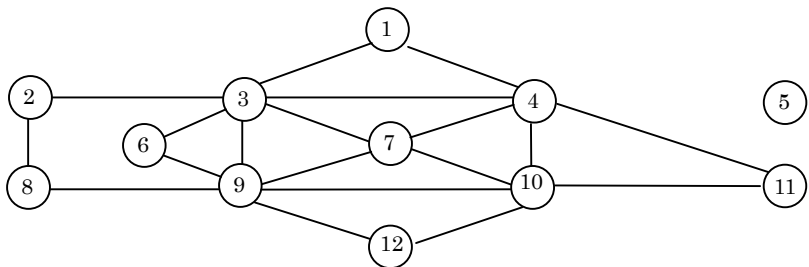



그림 9-40. 5, 4, 10, 5 다음의 나머지그라프

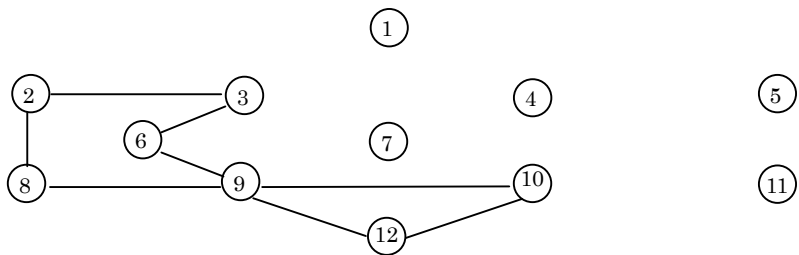



그림 9-41. 경로 5,4,1,3,7,4,11,10,11,9,3,4,10,5 다음의 그라프

남아 있는 그라프를 그림 9-42에 보여 주었다. 이 그라프상에서 순환되지 않는 변을 가지는 다음정점은 9이며 알고리듬은 회로 9, 12, 10, 9를 찾는다. 이것이 현재 경로에 첨부되면 5, 4, 1, 3, 2, 8, 9, 12, 10, 9, 6, 3, 7, 4, 11, 10, 7, 9, 3, 4, 10, 5의 회로가 얻어 진다. 이 변들이 모두 순환됨으로써 알고리듬은 오일레르회로를 가지고 끝낸다.

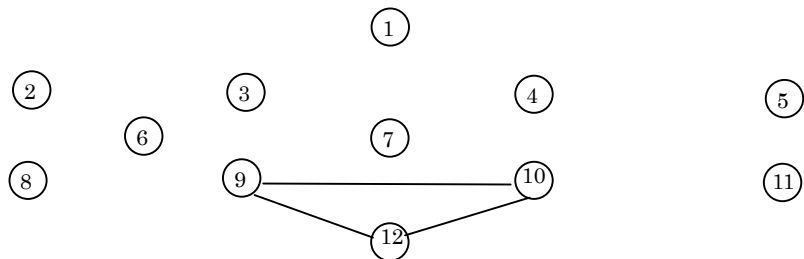



그림 9-42. 경로 5, 4, 1, 3, 2, 8, 9, 6, 3, 7, 4, 11, 10, 7, 9, 3, 4, 10, 5다음의 나머지그라프

이 알고리듬이 효률적인것으로 되기 위하여서는 알맞는 자료구조를 리용해야 한다. 그 방법에 대하여서는 일부 륜곽을 주고 그 실현을 련습문제로서 남겨 놓는다. 두 순환경로들을 간단히 련결하기 위하여 경로는 련결목록으로서 관리되여야 한다. 린접목록의 련속적인 주사를 피하기 위하여 매개 린접목록에 관하여 주사되는 마지막변에 대한 지적자를 보존해야 한다. 경로가 이어 지면 다음번 깊이우선탐색을 진행하기 위하여 그 어느것으로부터 새로운 정점에 관한 탐색은 정착점의 시작에서 시작해야 한다. 이것은 정점탐색단계에서 수행되는 전체 처리가 알고리듬의 처리과정에 $O(|E|)$이라는것을 담보한다. 알맞는 자료구조를 가지면 알고리듬의 실행시간은 $O(|E|+|V|)$로 된다.

가장 류사한 문제는 무방향그라프에서 모든 정점을 방문하는 단순한 순환을 찾는것이다. 이것은 **하밀톤순환문제**(*Hamiltonian cycle problem*)로 알려 져 있다. 이것이 오일레르회로문제와 거의 동일한것이라고 하지만 그에 관한 효률적인 알고리듬은 알려 져 있지 않다. 이 문제를 제9장 제7절에서 다시 본다.

4. 방향그라프

무방향그라프와 같은 전략을 리용하면 방향그라프는 깊이우선탐색을 리용하여 선형시간내에 순회할수 있다. 그라프가 강하게 련결되여 있지 않다면 임의의 매듭에서 시작하는 깊이우선탐색은 모든 매듭을 방문하지 못할수 있다. 이 경우에 모든 매듭을 방문할 때까지 어떤 표식기호가 붙지 않은 매듭에서 시작하여 반복적으로 깊이우선탐색을 수행한다. 실례로 그림 9-43에서 방향그라프를 고찰하자.

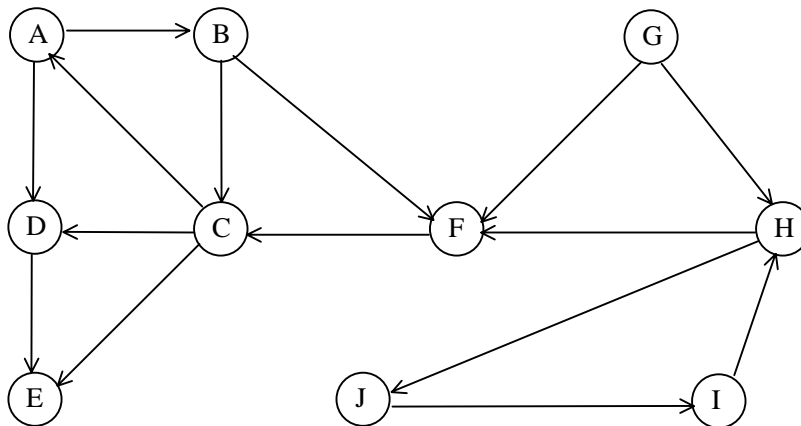



그림 9-43. 방향그라프

먼저 정점 *B*에서 깊이우선탐색을 시작한다. 이것은 정점 *B, C, A, D, E, F*를 방문한다. 그다음 방문하지 않은 어떤 정점에서 다시 시작한다. *H*에서 시작하고 *I*와 *J*를 방문한다.

마감에 G에 시작하며 이것은 방문하여야 할 마지막정점이다. 대응하는 깊이우선탐색나무를 그림 9-44에서 보여 주었다.

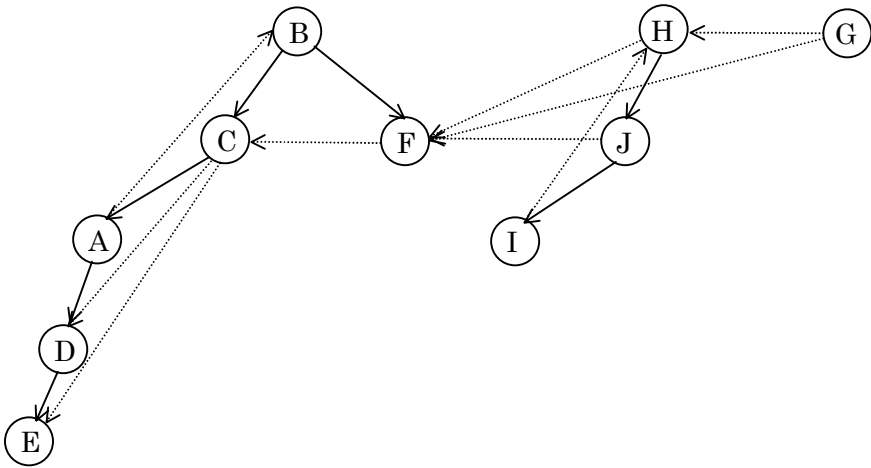



그림 9-44. 이전그라프의 깊이우선탐색

깊이우선생성수림에서 점선화살표는 고찰시에 이미 w에 표시기호를 붙인 변(v, w)이다. 무방향그라프에서 이것들은 항상 **뒤변**이지만 새로운 정점에로 인도하지 않는 3가지 형태의 변들이 있다. 먼저 (A, B)와 (I, H)와 같은 뒤변들이 있다. 또한 나무매듭으로부터 자식에로 유도하는 (C, D)와 (C, E)와 같은 **앞변**(*forward edges*)들이 있는데 그것은 나무매듭들에서 자식까지 유도한다. 마감에 직접 관련되여 있지 않는 2개의 나무매듭들을 련결하는 (F, C), (G, F)와 같은 **교차변**(*cross edges*)이 있다. 깊이우선탐색수림들은 일반적으로 자식들에 의하여 그려 지며 새로운 나무들이 왼쪽으로부터 오른쪽에로 수림에 첨부된다. 이러한 방식으로 그려 진 방향그라프의 깊이우선탐색에서 교차변은 항상 오른쪽으로부터 왼쪽으로 향한다.

깊이우선탐색을 리용하는 일부 알고리듬은 3가지 형태의 비나무변들을 식별하여야 한다. 이것은 깊이우선탐색이 실행되는것과 함께 검사하는것이 쉬우며 그것을 련습문제로서 남겨 놓는다.

깊이우선탐색의 한가지 리용은 방향그라프가 순환인가 비순환인가를 검사하는것이다(그의 그라프는 뒤변들을 가지므로 비순환이 아니다.). 규칙은 방향그라프가 뒤변을 가지지 않을 때에만 비순환이라는것이다. 위상학적정렬은 그라프가 비순환인가를 결정하는데 리용될수도 있다. 위상학적정렬을 진행하는 다른 한가지 방법은 깊이우선생성수림의 후뿌리순회에 의하여 정점들의 위상학적번호 N, N-1,⋯,1을 할당하는것이다. 그라프가 비순환인 한 이 순서는 일정하다.

5. 강한 련결성분찾기

두번의 깊이우선탐색을 진행함으로써 방향그라프가 강하게 련결되였는가를 검사할수 있으며 강한련결그라프가 아니면 그 자체내에서 강하게 련결된 정점들의 부분모임을 실제로 생성할수 있다. 이것은 또한 하나의 깊이우선탐색으로써 수행할수 있지만 여기서 리용한 방법은 리해하기 훨씬 더 쉽다.

먼저 입력그라프 *G*에서 깊이우선탐색이 진행된다. *G*의 정점들은 깊이우선생성수림의 후뿌리순회에 의하여 번호가 붙여 지며 그 다음에 *G*의 모든 변들을 반전하여 G_r를 만든다. 그림 9-45의 그라프는 그림 9-43에서 보여 준 그라프 *G*에 관한 G_r를 나타내는데 여기에서 정점들은 그 번호로 보여 주었다.

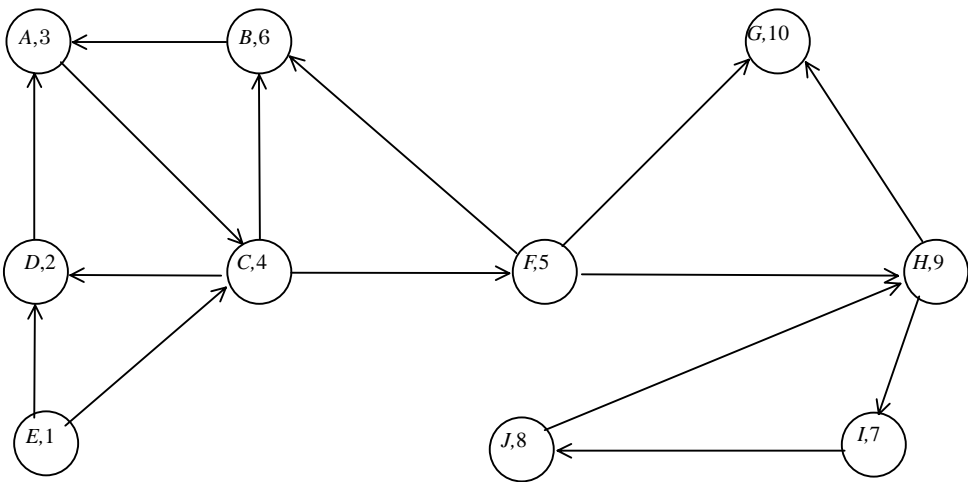



그림 9-45. G(그림 9-43)의 후뿌리순회로 번호매겨 진 Gr

알고리듬은 G_r상에서 깊이우선탐색을 진행함으로써 완성되며 항상 가장 높은 번호가 붙은 정점에서 새로운 깊이우선탐색을 시작한다. 따라서 번호가 10인 정점 *G*에서 G_r에 대한 깊이우선탐색을 시작한다. 이것은 그 어디에도 향하지 않으므로 다음번 탐색은 *H*에서 시작된다. 이것은 *I*와 J를 방문한다. 다음번 호출은 *B*에서 시작하여 *A, C, F*를 방문한다. 이 다음의 호출은 dfs(D)이며 마감에 dfs(E)이다. 결과로 되는 깊이우선생성수림을 그림 9-46에서 보여 주었다.

깊이우선생성나무에서 나무의 매개는 강하게 련결된 구성요소를 형성한다. 따라서 우의 실례에서 강하게 련결된 구성요소는 {*G*}, {*H, I, J*}, {*B, A, C, F*}, {*D*}, {*E*}이다.

이 알고리듬의 처리과정을 보기 위하여 먼저 두개의 정점 *v*와 *w*가 강한련결성분이라면 그때 초기그라프 *G* 에서 *v*로부터 *w*에로, *w*로부터 *v*에로의 경로들이 존재하며 이것들

은 그라프 G_r에서도 마찬가지이다. 이제 두개의 정점 v와 w가 G_r의 같은 깊이우선생성나무에 있지 않다면 명백하게 그것들은 같은 강한련결성분내에 있을수 없다. 이 알고리듬이 동작하는가를 증명하기 위하여 두개의 정점 v와 w가 G_r의 같은 깊이우선생성나무에 없다면 v로부터 w에로와 w로부터 v에로 경로가 있어야 한다는것을 증명하여야 한다. 동시에 x가 v를 포함하는 G_r의 깊이우선생성나무의 뿌리이면 x로부터 v에로와 v로부터 x에로 경로가 있다는것을 확인할수 있다.

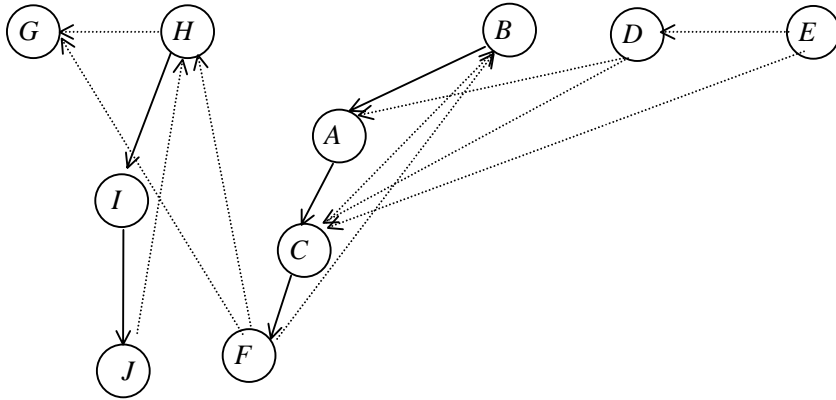



그림 9-46. G_r의 깊이우선탐색-강하게 련결된 구성요소는 {G}{H, I, J}{B, A, C, F}{D}{E}이다.

w에 같은 론리를 적용하면 x로부터 w에로와 w로부터 x에로 경로를 준다. 이 경로들은 v로부터 w에로, w로부터 v에로 경로를 암시한다.

v는 G_r의 깊이우선생성나무에서 x의 자식이므로 G_r에서 x로부터 v에로 경로가 있으며 따라서 G에서 v로부터 x에로 경로가 있다. 더우기 x가 뿌리이므로 x는 처음의 깊이우선탐색으로부터 보다 높은 후뿌리순회번호를 가진다. 그러므로 우선 깊이우선탐색과정에 v를 처리하는 모든 동작은 x에서 동작이 완성되기전에 완성된다. v로부터 x에로 하나의 경로가 있으므로 v는 G에 관한 생성나무에서 x의 자식이 되여야 하며 다른 한편 v는 x다음에 끝난다. 이것은 G에서 x로부터 v에로 경로를 암시하며 증명을 완성한다.

제7절. NP-완전성의 소개

이 장에서 광범한 종류의 그라프리론문제들에 대한 해결을 고찰하였다. 이 모든 문제들은 비결정적인 다항식시간(Nondeterministic polynomial time)을 가지며 망흐름문제를 제외하고 실행시간은 선형적이거나 약간 덜 선형적인것으로서 $O(|E|log|E|)$이다. 일부 문제들에 대하여 약간의 변화가 처음의 문제보다 더 힘들어 보인다고 언급하였다.

모든 변을 정확히 한번만 순회하는 경로를 찾는 오일레르순회문제는 선형시간내에 풀수 있다. 하밀톤순환문제는 모든 정점을 포함하는 간단한 순환을 요구한다. 이 문제에 대한 비선형알고리듬들은 알려 져 있지 않다.

또한 방향그라프에서 단일원천무게없는최단경로문제도 선형시간내에 해결할수 있다. 가장 긴 단순경로문제에 대응하는 비선형알고리듬들은 알려 져 있지 않다.

이러한 문제변종들의 상태는 이미 서술한것보다 실제로 더 나쁘다. 이러한 변종들에 대한 선형알고리듬들은 전혀 알려지지 않았을뿐아니라 다항식시간에 실행되는 알고리듬들은 알려 진것이 없다. 이러한 문제에 대하여 가장 잘 알려 진 알고리듬들은 어떠한 입력값들에 대하여 지수적인 시간을 가진다.

이 절에서는 이 문제에 대하여 간단히 론의한다. 이 문제는 복잡하므로 략하여 언급한다. 때문에 ***NP***-완전성에 대한 론의는 여러곳에서 좀 부정확할수 있다.여기서는 대체로 동등한 복잡성을 가지는 주요한 문제들이 수많이 존재한다는것을 알게 될것이다. 이러한 문제들을 ***NP*−완전성문제**(*complete problem*)라고 한다. 이 *NP*−완전성문제들의 정확한 복잡도는 앞으로 결정되여야 하며 리론적인 콤퓨터과학에서 제일 잘 알려 진 문제로 남아있다. 이 모든 문제들은 다항식시간풀기를 가지고 있기도 하나 그것들중 아무것도 해결되지 않고 있다.

1. 쉬운 문제와 어려운 문제

이 문제들을 분류할 때 첫 단계는 한계들을 검사하는것이다. 이미 많은 문제들이 선형시간내에 풀릴수 있다는것을 보았다. 또한 일련의 $O(logN)$실행시간을 보았는데 이것들은 일련의 전처리가 진행된다고 가정(이미 읽어 들인 입력이나 이미 구축된 자료구조와 같은)하거나 산수적실례들에서 나타난다. 실례로 *gcd*알고리듬은 M, N 두수에 대해 적용될 때 $O(logN)$시간을 가진다. 수자들은 각각 $logM$과 $logN$의 비트로 구성되므로 *gcd*알고리듬은 실지로 입력량이나 크기에서 선형적인 시간을 가진다. 따라서 실행시간을 측정할 때 실행시간을 입력량의 함수와 같은것으로서 취급한다. 일반적으로 선형적인 실행시간보다 더 좋은것을 기대할수 없다.

한편 실지 어려운 문제들이 일부 있다. 이 문제들은 너무 힘들어서 그에 대한 분석이 불가능하다. 그러나 이것은 절대적인 불가능을 의미하지 않으며 이 문제들은 앞으로 능히 해결할수 있다. 실수가 $x^2<0$이라는 풀이를 나타내기에 충분하지 않을 때 바로 사람은 콤퓨터가 그로하여 발생하는 모든 문제를 풀수 없다는것을 증명할수 있다. 이 《불가능한》문제들을 **결정불가능**(*undecidable*)문제라고 한다.

한가지 독특한 결정불가능문제는 《정지문제》이다. C++번역기가 문법적오유뿐아니라 모든 무한순환까지도 발견해 내는 특별한 특징을 가지도록 하는것이 가능한가? 이것

은 어려운 문제처럼 보이나 일부 사람들은 재능 있는 프로그람작성자들이 여기에 품을 들이면 기적을 일으킬수 있을것이라고 기대할수도 있다.

이 문제가 결정불가능하다는 직관적리유는 그러한 프로그람이 자기자체를 검사하는데 지나친 시간이 걸린다는것이다. 이러한 리유로 하여 이 문제들을 때때로《재귀적으로 결정불가능》이라고 한다.

무한순환고리검사프로그람이 씌여 질수 있다면 분명히 그자체를 검사하는데 사용될수도 있다. 그때에 *LOOP*라고 하는 프로그람을 만들어 낼수도 있을것이다. *LOOP*는 프로그람 *P*를 입력하고 그 자체에 대하여 *P*를 실행한다. 그자체에 대하여 실행될 때 *P*가 순환고리를 이룬다면 문장 Yes를 출력한다. *P*가 그자체에 대하여 실행될 때 완료되면 응당 처리하여야 할 문제는 No를 출력하는것이다. 그러한 처리대신에 *LOOP*가 무한순환으로 들어 가게 한다.

*LOOP*자체가 입력량으로 주어 질 때 어떤 일이 일어 나겠는가? *LOOP*가 정지되거나 정지되지 않는다. 문제는 이 두가지 가능성들이 문장《이 문장은 거짓말이다.》라는것과 같은 방식으로 모순에로 유도한다는것이다.

우에서의 정의에 의하면 *LOOP(P)*는 *P(P)*가 끝난다면 무한순환에 들어 간다. *P*=*LOOP*일 때 *P(P)*가 끝난다고 하자. 그러면 *LOOP*프로그람에 따라서 *LOOP(P)*는 무한순환에 들어갈 의무를 가진다. 따라서 무한순환을 끝내거나 무한순환에로 들어 가는 *LOOP*(*LOOP*)를 가져야 하는데 그것은 명백히 가능하지 않다. 한편 *P*=*LOOP*일 때 *P(P)*는 무한순환에 들어 간다고 가정하자. 그때 *LOOP(P)*는 완료되여야 하며 같은 모순에 봉착한다. 이와 같이 *LOOP*프로그람은 도저히 존재할수 없다.

2. NP 클라스

비결정문제들에 대한 문제로부터 제기되는 일련의 단계들은 *NP*클라스문제이다. *NP*는 《**결정할수 없는 다항식시간**(*nondeterminstic polynomial-time*)》을 상징한다. 결정론적인 기계는 매 시점에서 제때에 그 명령을 실행하고 있다. 그 명령에 의존하여 그 다음에는 어떤 다음명령에로 가는데 그다음 명령이라는것은 오직 하나이다. 비결정론적인 기계는 다음 단계들에 대한 선택을 가지고 있다. 자기가 원하는 그 어떤것이건 선택하는것은 자유로운데 이 단계들중의 하나가 풀이로 된다면 그것은 항상 정확한 처리를 선택할것이다. 이처럼 비결정론적인 기계는 매우 훌륭한 (최적의) 추측능력을 가지고 있다. 이것은 누구도 도저히 비결정론적인 콤퓨터를 만들수 없고 사용자의 표준콤퓨터에 대한 놀라운 향상으로 되여 보이기때문에 허항한 모형처럼 보인다. 모든 문제는 현재 사소한것처럼 보일수 있다. 여기에서 비결정론은 대단히 유용한 리론적구성개념이라는것을 알게 될것이다. 더군다나 비결정론은 사람이 생각하듯이 그렇게 강력한것은 아니다. 실례로 결정불가능한 문제들은 비결정론이 허락된다 하더라도 여전히 결정불가능하다.

문제가 *NP*인가를 검사하는 간단한 방법은 Yes/No물음으로 그 문제를 표현하는것이다. 그 문제는 다항식시간내에 그 어떤 "Yes"가 정확하다는것을 증명할수 있다면 *NP*이다. 프로그람은 항상 옳은 선택을 하므로 "No"경우에 대해 걱정하지 말아야 한다. 따라서 하밀톤순환경로문제에서 "Yes"경우는 모든 정점들을 포함하는 그라프의 어떤 간단한 회로가 된다. 이것은 경로가 주어 지면 그것이 실지로 하밀톤순환경로라는것을 검사하는 간단한 문제이므로 *NP*문제에 속한다. ≪길이>K인 단순경로가 있는가?≫와 같은 적당히 표현된 물음들은 또한 쉽게 검사될수 있고 또 *NP*이다. 이 속성을 만족시키는 경로는 명백하게 검사될수 있다.

*NP*클라스는 다항식시간풀이를 가지는 모든 문제들을 포함하는데 그것은 이 풀이가 명백한 검사를 제공하기때문이다. 사람은 령으로부터 시작해 나가는것보다는 어떠한 결과를 검사하는것이 더 쉬우므로 다항식시간풀이를 가지지 않는 *NP*에 속하는 문제들이 있을것이라고 기대한다. 지금까지 이러한 문제가 나타난것은 아예 없으며 그래서 전문가들에 의해 십중팔구 고찰되지는 않았지만 비결정론이 그다지 중요한 개선은 아니라고 생각할수 있다. 문제는 지수적인 아래한계들을 증명하는것이 매우 어려운 과제라는것이다. 정렬이 $\Omega(N\log N)$의 비교를 요구한다는것을 보여 주는데 리용하는 정보리론경계기술은 결정나무들이 거의 충분히 크지 않기때문에 그 과제에 적절한것으로 보이지 않는다.

또한 모든 결정가능한 문제들이 *NP*에 속하지 않는다는것을 주의하여야 한다. 그라프가 하밀톤순환경로를 가지지 않는가 하는것을 결정하는 문제를 고찰하자. 그라프가 하밀톤순환경로를 가진다는것을 증명하는것은 비교적 간단한 문제이다. 바로 그 문제를 제시하여야 한다. 누구도 다항식시간내에 그라프가 하밀톤순환경로를 가지지 않는다는것을 증명하는 방법을 제시하지 못하였다. 여기서는 모든 순환경로들을 렬거하고 그것들을 하나씩 검사해야 한다. 이처럼 비하밀톤주기문제는 *NP*라고는 알려 져 있지 않다.

3. NP -완전성문제

*NP*문제들중에는 최대의 난점을 포함하고 있는 ***NP*−완전성문제**라고 알려 진 부분모임이 있다. *NP*의 어떤 문제가 그에 대해 다항식적으로 변형될수 있는 속성을 가진다.

문제 P_1는 다음과 같이 P_2로 변형될수 있다. P_1의 어떤 객체가 P_2의 객체로 변형될수 있도록 넘기기를 준비한다. P_2를 풀고 다시 그 대답을 본래의것에로 돌려 넘기시오. 실례로 수자들은 전자수산기에 10진수로 들어 간다. 10진수들은 2진수로 변환되여 모든 계산은 2진법으로 진행된다. 그다음 최종결과를 표시하기 위해 다시 10진수로 변환된다. P_1가 다항식적으로 변형해 지도록 하기 위해서는 모든 변형과 관련된 작업들이 다항식시간내에 진행되여야 한다.

*NP*완전성문제들이 가장 어려운 *NP*문제들이라는 리유는 *NP*문제가 오직 다항식적으로 부가되는것을 가지는 임의의 *NP*문제의 부분루틴으로서 주요하게 사용될수 있다는것

이다. 따라서 어떤 NP완전성문제가 다항식시간풀이를 가진다면 그때 NP인 매 문제는 다항식시간풀이를 가져야 한다. 이것은 NP완전성문제가 모든 NP문제들가운데서 가장 어려운 문제라는것을 의미한다.

NP완전성문제 P_1가 있다고 하자. P_2은 NP라고 알려 져 있다고 가정하시오. 더우기 P_1는 P_2로 다항식적으로 변형되며 P_2을 리용하여 P_1를 오직 다항식시간조건으로 풀수 있다고 하자. P_1는 NP완전성이므로 NP에 속하는 문제는 P_1에 대해 다항식적으로 변형된다. 다항식들에 대한 종결속성을 적용함으로써 NP의 매 문제는 P_2로 다항식적으로 변환가능하다는것을 알수 있는바 문제를 P_1로 변형시키고 그다음 P_1를 P_2로 변형시킨다. 따라서 P_2은 NP완전성이다.

실례로 이미 하밀톤순환경로문제가 NP완전성이라는것을 알고 있다고 가정하자. 순회판매원문제(traveling salesman problem)는 다음과 같다.

순회판매원문제:

변의 무게와 하나의 옹근수 K를 가지는 완전그라프 $G=(V,E)$가 주어 졌을 때 모든 정점들을 방문하면서 총 원가≦K인 단순순환경로가 있는가?

이 문제는 $|V|(|V|-1)/2$개의 변들이 존재하고 그라프가 무게를 가지기때문에 하밀톤순환경로문제와는 다르다. 이 문제는 많은 중요한 응용프로그람들을 가진다. 실례로 인쇄회로기판은 소편, 저항기와 다른 전기적요소들이 놓일수 있도록 뚫려진 구멍만을 가져야 한다. 이것은 기계적으로 진행된다. 구멍뚫기는 빠른 조작이다. 시간소비단계는 구멍뚫기기구를 위치조종하는데 있다. 위치조종에 요구되는 시간은 구멍사이를 움직이는 거리에 관계된다. 모든 구멍을 뚫으려고 하므로(그리고 다음기판의 시작위치로 돌아 간다.) 움직이는데 소비되는 전체 시간량은 될수록 적게 한다. 이것은 여기서 론의하려는 순회판매원문제와 류사하다.

순회판매원문제는 NP완전성이다. 다항식시간내에 풀이가 검사될수 있는가를 고찰함으로써 그것이 명백히 NP라고 보는것이 더 쉽다. NP완전성이라는것을 보여 주기 위하여 우리는 다항식적으로 하밀톤함수주기문제를 변형시킨다. 이렇게 하기 위하여 새 그라프 G'를 구축한다. G'는 G와 같은 정점들을 가진다. G'에서 매개 변 (v,w)는 $(v,w)\in G$이면 무게 1을 가지며 그외에는 2를 가진다. $K=|V|$라고 선택한다. 그림 9-47을 보시오.

G'가 전체무게 $|V|$를 가지는 순회판매원경로를 가질 때에만 G는 하밀톤순환경로를 가진다는것을 확증하기 쉽다.

현재 NP완전성으로 알려 진 어떤 새로운 문제가 NP-완전성이라는것을 증명하기 위하여서는 먼저 NP라는것을 증명하여야 하며 그다음 그것을 적당한 NP완전성문제로 변형하여야 한다. 순회판매원문제에로의 변형이 어느 정도 직접적이지만 실제로 많은 변형들이 포함되며 약간의 기묘한 구축이 요구된다. 일반적으로 여러가지 각이한 NP완전성문

제들은 실제로 변형하기 전에 고찰된다. 여기서는 일반적인 개념들에만 관심을 가지므로 그 이상의 변형들은 보여 주지 않는데 참고서들을 리용할수 있다.

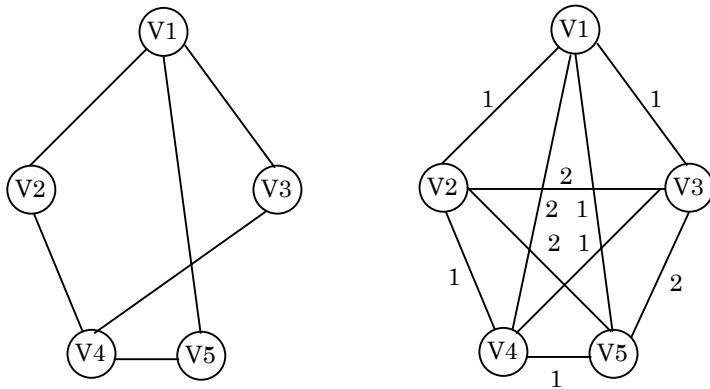



그림 9-47. 순회판매원문제로 전환된 하밀톤순환경로문제

여기에서 첫 *NP*완전성문제가 어떻게 증명되는가를 이상하게 여길수도 있다. 문제가 *NP*완전성이라는것을 증명하는것은 그것을 다른 *NP*완전성문제로부터 그것을 변형할것을 요구하므로 이 방법이 처리할수 없는 그어떤 *NP*완전성문제가 있어야 한다. *NP*완전성으로 증명된 첫 문제는 만족성(satisfiability)문제였다. 만족성문제는 론리식을 입력으로 가지며 그 식이 true값으로 변수에 값주기를 하는가를 검사한다.

론리식을 평가하고 결과가 true인가를 검사하는것은 쉬우므로 만족성문제는 명백히 *NP*이다. 1971년에 쿠크(cook)는 모든 *NP*문제들이 만족성으로 변형될수 있다는것을 직접 증명함으로써 만족성이 *NP*−완전성이라는것을 보여 주었다. 이를 위하여 그는 *NP*인 매 문제에 대한 하나의 잘 알려 진 사실을 리용하였는데 그것은 *NP*인 매 문제는 비결정론적인 콤퓨터에 의해 다항식시간에 풀수 있다는것이다. 콤퓨터에 대한 고전적인 모형을 **튜링기계**(*Turing machine*)라고 한다. 쿠크는 이 기계의 동작이 아주 복잡하고 길지만 여전히 다항식적인 론리식을 가지고 모의할수 있는 방법을 보여 주었다. 이 론리식은 튜링기계에 의해 실행되고 있는 프로그람이 자기의 입력에 대하여 《예》대답을 준다면 참으로 된다.

만족성이 *NP*완전성이라고 보여 진이래 가장 대표적인 일부 문제들을 포함하는 새로운 *NP*완전성문제들도 *NP*완전성으로 증명된다.

이미 시험해 본 만족성외에 하밀톤회로문제, 순회판매원문제, 최장경로문제들에서 론의하지 않은 잘 알려 진 *NP*완전성문제는 큰 상자채우기문제, 배낭채우기, 그라프채색문제, clique문제들이다. 그 목록은 아주 방대하며 조작체계(일정작성과 안전성), 자료기지체계들, 작전연구, 론리 그리고 특히 그라프리론에 대한 문제들을 포함한다.

요약

이 장에서는 실제적인 문제들을 모형화하는데 그라프를 어떻게 리용하겠는가 하는것을 보았다. 형태적으로 아주 성긴 대부분의 그라프들은 그것들을 실현하는데 리용되는 자료구조들에 관심을 두어야 한다.

여기서는 또한 효과적으로 해결할수 없는것처럼 보이는 문제들에 대해서 고찰하였다. 제10장에서는 이 문제들을 처리하는 몇가지 수법들을 론의하게 된다.

련습문제

9-1. 그림 9-48에서 그라프의 위상학적순서를 찾으시오.

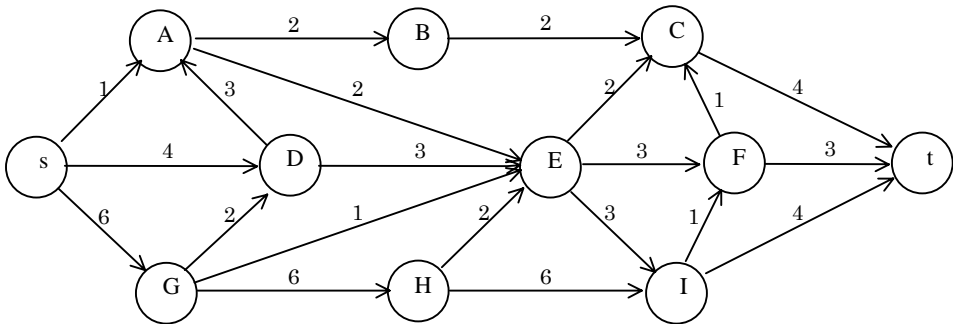



그림 9-48. 련습문제 9-1과 9-11에 리용되는 그라프

9-2. 제9장 제1절의 위상학적정렬알고리듬에서 대기렬대신에 탄창이 사용된다면 순서결과가 달라지는가? 왜 탄창자료구조가 《더 좋은》대답을 주는가?

9-3. 그라프에 대한 위상학적정렬을 진행하는 프로그람을 작성하시오.

9-4. 린접행렬은 다만 표준쌍방향순환을 사용하여 초기화하기 위해 $O(|V|^2)$을 요구한다. 린접행렬에 그라프를 보관하되(변의 존재검사가 $O(1)$이도록) 그러나 2차실행시간을 피하는 방법을 생각해 내시오.

9-5. ㄱ. 그림 9-49의 그라프에서 A로부터 모든 정점들에로 향하는 최단경로를 찾아 내시오.

ㄴ. 그림 9-49의 그라프 B로부터 다른 모든 정점들에로 향하는 무게없는최단경로를 찾아 내시오.

9-6. d-더미(제6장 제5절)로 실행될 때 최악의 경우 딕스트라알고리듬의 실행시간은 얼마인가?

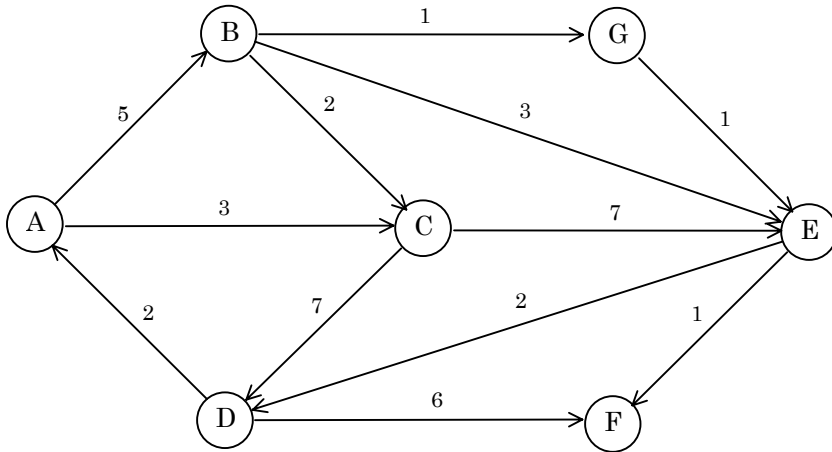



그림 9-49. 련습문제 9-5의 그라프

9-7. ㄱ. 딕스트라알고리듬이 부의 무게변들은 있으나 부값순환경로는 없다는것에 대한 잘못된 해답을 주는 실례를 드시오.

ㄴ. 제9장 제3절 3에 보여 준 무게붙은최단경로알고리듬이 부무게변들이 있지만 부값순환이 없을 때 동작하며 이 알고리듬의 실행시간이 $O(|E|\cdot|V|)$ 이라는것을 증명하시오.

***9-8.** 그라프의 모든 변의 무게가 1과 |E|사이의 옹근수라고 가정하시오. 딕스트라알고리듬이 얼마나 빨리 실행될수 있는가?

9-9. 단일원천최단경로문제를 푸는 프로그람을 작성하시오.

9-10. ㄱ. v로부터 w에 이르는 각이한 최소경로들의 총 개수를 계산하도록 디직스트라알고리듬을 수정하는 방법을 설명하시오.

ㄴ. v로부터 w에 이르는 하나 이상의 최단경로가 있다면 변들의 최소개수를 가지고 하나의 경로가 선택되도록 딕스트라알고리듬을 수정하는 방법을 설명하시오.

9-11. 그림 9-48의 망에서 최대흐름을 찾으시오.

9-12. $G=(V,E)$는 나무, S는 그 뿌리인데 여기에 정점 t와 G의 모든 잎으로부터 t에로 향하는 무한한 능력의 변들을 추가한다고 하시오. S로부터 t에로 흐르는 최대흐름을 찾는 선형시간알고리듬을 작성하시오.

9-13. 2진그라프 $G=(V,E)$는 V가 두개의 부분모임 V_1와 V_2로 분리될수 있고 동일한 부분모임의 정점들을 가지는 변이 하나도 없는 그라프이다.

ㄱ. 그라프가 2진그라프인가 아닌가를 결정하는 선형알고리듬을 작성하시오.

ㄴ. 갈라 지는 맞물림문제는 하나이상의 변에 포함되는 정점이 하나도 없는 E의 가장 큰 부분모임 E'를 찾아 내는것이다. 4개 변의 한가지 맞물림

(점선들로 표시된)을 그림 9-50에 보여 주었다. 5개 변의 맞물림이 하나 있는데 그것은 최대값이다. 갈라 지는 맞물림문제가 다음의 문제를 푸는데 어떻게 사용될수 있는가를 보여 주시오.선생들의 모임, 과목들의 모임, 매 선생이 가르칠 자격이 있는 과목들의 목록, 만일 선생이 한과목이상을 가르칠 필요가 전혀 없고 또 오직 한 선생이 주어 진 한 과목을 가르칠수 있다면 제공될수 있는 과목들의 최대수는 얼마인가?

ㄷ. 망흐름문제가 갈라 진 맞물림문제를 푸는데 사용될수 있다는것을 보여 주시오.

ㄹ. ㄴ을 분할하는 풀이의 시간복잡도는 얼마인가?

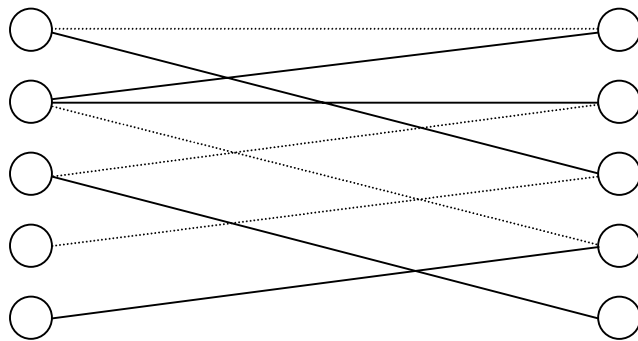



그림 9-50. 2진그라프

9-14. 최대흐름을 허락하는 증가경로를 찾아 내는 알고리듬을 작성하시오.

9-15. ㄱ. 프림과 크루스칼의 알고리듬들을 리용하여 그림 9-51의 그라프에 대한 최소생성나무를 찾아내시오.

ㄴ. 이 최소생성나무는 유일한가? 왜 그런가?

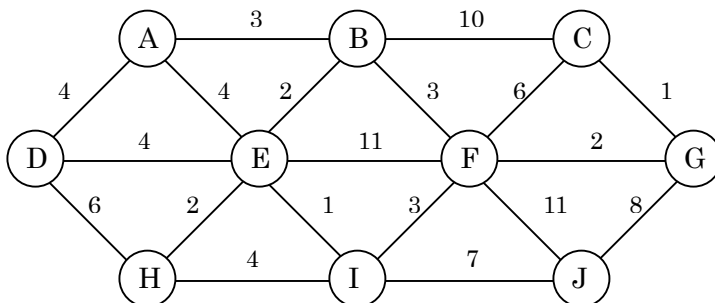



그림 9-51. 련습문제 9-15에 리용되는 그라프

9-16. 프림의 알고리듬이나 크루스칼의 알고리듬은 부의 무게변이 있을 때에도 동작하는가?

9-17. V정점들의 그라프는 V^{V-2}최소생성나무들을 가질수 있다는것을 증명하시오.

9-18. 크루스칼의 알고리듬을 실현하는 프로그람을 작성하시오.

9-19. 그라프의 모든 변들이 1과 $|E|$사이의 무게를 가지고 있다면 최소생성나무는 얼마나 빨리 계산될수 있는가?

9-20. 최대생성나무를 구하는 알고리듬을 작성하시오. 이것은 최소생성나무를 구하기보다 더 힘든가?

9-21. 그림 9-52의 그라프에서 분리점들을 모두 구하시오. 깊이우선생성나무와 매 정점의 *Num*과 *Low*값들을 보여 주시오.

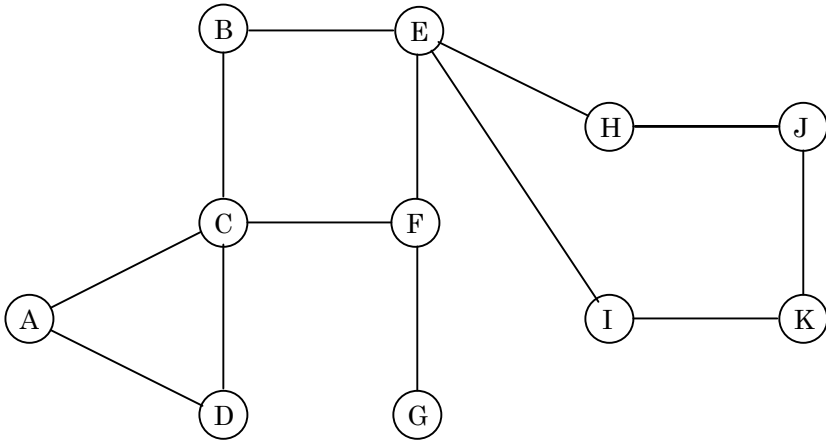



그림 9-52. 련습문제 9-21에 리용되는 그라프

9-22. 분리점들을 구하는 알고리듬이 동작한다는것을 증명하시오.

9-23. ㄱ. 그라프가 비순환이 되도록 무방향그라프로부터 제거되여야 할 변들의 최소수를 구하는 알고리듬을 작성하시오.

*ㄴ. 이 문제는 방향그라프에 대하여 *NP*-완전이라는것을 증명하시오.

9-24. 방향그라프의 깊이우선생성수림에서 모든 가로지름변들은 오른쪽에서 왼쪽으로 간다는것을 증명하시오.

9-25. 방향그라프의 깊이우선생성수림의 변 (v, w)이 나무변, 뒤변, 교차변 혹은 앞변인가를 결정하는 알고리듬을 작성하시오.

9-26. 그림 9-53의 그라프에서 강한 련결성분들을 찾아 내시오.

9-27. 2중그라프에서 강한 련결성분들을 찾아 내는 프로그람을 작성하시오.

***9-28.** 오직 한번의 깊이우선탐색으로 강한 련결성분들을 찾아 내는 알고리듬을 작성하시오. 쌍련결알고리듬과 류사한 알고리듬을 리용하시오.

9-29. 그라프 G의 **쌍련결성분**(*biconnected components*)들은 매개 변들의 모임에 의해 형성된 그라프가 쌍련결된것과 같은 모임들에 들어 있는 변들의 부분이

다. 분리점들대신에 쌍련결된 성분들을 찾아 내도록 프로그람 9-13의 알고리듬을 수정하시오.

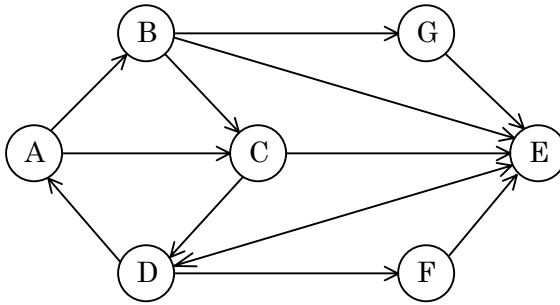



그림 9-53. 련습문제 9-26에 리용되는 그라프

9-30. 무방향그라프에 대해 너비우선탐색을 진행하고 너비우선생성나무를 만든다고 가정하시오. 나무의 모든 변들이 다른 한쪽의 나무변들이거나 아니면 교차변들이라는것을 증명하시오.

9-31. 무방향련결그라프에서 모든 변들이 매 방향으로 정확히 한번만 통과 해나가는 경로를 찾는 알고리듬을 작성하시오.

9-32. ㄱ. 오일레르순회가 존재하는 그라프에서 오일레르회로순회경로를 찾아 내는 프로그람을 작성하시오.

ㄴ. 오일레르순회가 존재하는 그라프에서 오일레르순회를 찾아 내는 프로그람을 작성하시오.

9-33. 방향그라프의 오일레르회로는 매 변이 정확히 한번 방문되는 하나의 순환경로이다.

*ㄱ. 방향그라프는 오일레르회로가 강하게 련결되고 매 정점이 동일한 입력도수와 출력도수를 가지며 또 오직 가진다면 그 방향그라프는 오일레르회로를 가진다는것을 증명하시오.

*ㄴ. 방향그라프에서 그 그라프의 오일레르회로를 발견하는 선형시간알고리듬을 작성하시오.

9-34. ㄱ. 오일레르회로문제에 대한 다음의 풀이를 고찰하시오. 즉 그라프가 쌍련결되였다고 가정하시오. 변들을 마지막모임으로서만 돌려 보내도록 깊이우선탐색을 진행하시오. 그라프가 쌍련결된것이 아니라면 쌍련결된 구성요소들에 대하여 재귀적으로 알고리듬을 적용하시오.

ㄴ. 변들을 돌려 보낼 때 가장 가까운 선조에게도 뒤변을 돌려 보낸다고 가정하시오. 알고리듬이 동작하는가?

9-35. 평면그라프(planar graph)는 임의의 두개의 변들이 교차하지 않고 평면에 그려질수 있는 그라프이다.

*ㄱ. 그림 9-54의 그라프들중 그 어느것이든 평탄하지 않다는것을 증명하시오.

ㄴ. 평면그라프에서 5개이상이 아닌 매듭들에 련결되는 일부 정점들이 존재해야 한다는것을 보여 주시오.

**ㄷ. 평면그라프에서 $|E| \leqq 3|V|-6$이라는것을 보여 주시오.

9-36. 다중그라프(multigraph)는 정점쌍들사이에 여러개의 변들이 존재하는 그라프이다. 이 장의 알고리듬들중 어느것이 다중그라프에 대해 수정함이 없이 작업하는가? 다중그라프에 대해서 작업하려면 어떠한 수정이 필요한가?

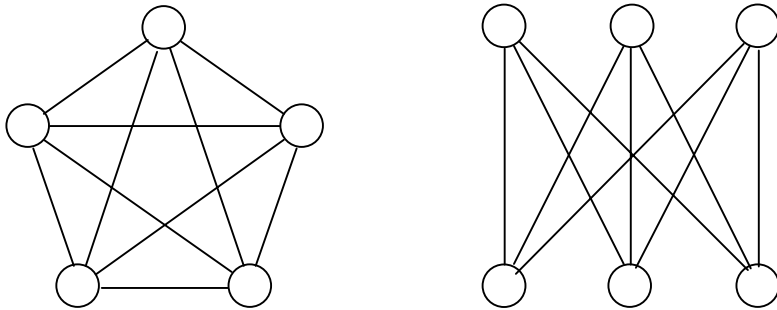



그림 9-54. 련습문제 9-35에 리용되는 그라프

***9-37.** $G=(V,E)$는 무방향그라프라고 가정하시오. 깊이우선탐색을 리용하여 G의 매개 변을 결과그라프가 강하게 련결되는 방향그라프로 전환시키거나 그렇게 할수 없다는것을 결정하는 선형알고리듬을 작성하시오.

9-38. N개의 가지들로 된 모임이 주어 졌다고 하자. 이때 가지들은 일부 배치에서 서로의 꼭대기에 놓여 있다. 매 가지는 자기의 두 끝점들에 의해 규정되는데 매 끝점은 자기의 x, y, z좌표를 주는 3개의 순서화된 쌍이다. 그러나 가지는 수직이 아니다. 하나의 가지는 그 꼭대기에 가지가 없다면 그때만은 잡을수 있다.

ㄱ. 두개의 가지 a와 b를 잡되 a가 우인가 아래인가 또는 b에 관계 없는가를 알려 주는 루틴을 작성하는 방법을 설명하시오(이것은 그라프리론으로 하는것이 아니다.).

ㄴ. 모든 가지들을 잡아 낼수 있는가 또 그렇다면 이것을 진행하는 가지잘라내기렬을 제공하는 알고리듬을 작성하시오.

9-39. 매 정점이 k개의 색갈중 하나로 주어 질수 있고 동일한 색갈의 정점들을 련결하는 변들이 하나도 없다면 k색갈성이다. 그라프가 그 색갈성인가를 검사하는 선형시간알고리듬을 작성하시오. 그라프들은 린접표형식으로 보관된다고 가정하되 필요한 추가적인 자료구조들을 규정해야 한다.

9-40. 임의의 무방향그라프의 변들의 거의 3/4을 공동으로 포괄하는 $[V/2]$정점들을 발견하는 다항시간알고리듬을 작성하시오.

9-41. 그라프가 비순환이 아니라면 알고리듬이 일부 순환을 출력해 내도록 위상학적정렬알고리듬을 수정하는 방법을 보여 주시오. 깊이우선탐색을 리용하지 않아도 된다.

9-42. G가 N개의 정점들로 된 방향그라프라고 하자. $s \neq v$인 V의 매 v에 대하여 하나의 변 (v,s)가 있고 (s,v)형식의 변들이 없다면 정점 s는 **홈**(*sink*)이라고 한다. G가 $n \times n$린접행렬로 주어 진다고 가정하고 G가 홈을 가지는가 안가지는가를 결정하는 $O(N)$인 알고리듬을 작성하시오.

9-43. 정점과 그에 들어 가는 변들이 나무로부터 제거될 때 부분나무들의 집합이 남아 있게 된다. N정점나무에서 그의 제거가 $N/2$이상의 정점을 가진 부분나무는 전혀 남기지 않는 그러한 정점을 발견하는 선형시간알고리듬을 작성하시오.

9-44. 무방향순환그라프(즉 나무)의 무게 없는 최대길이경로를 결정하는 선형시간알고리듬을 작성하시오.

9-45. 일부 4각형칸들이 검은 원들에 의해 점령되는 N-N격자칸무늬를 고찰하시오. 두개의 4각형칸들이 공통변을 공유하면 같은 그루빠에 속한다. 그림 9-55에서 보면 4개가 점령된 4각형들은 한개, 두개가 점령된 4각형들은 3개, 개별적으로 점령된 4각형들이 2개 있다. 격자칸무늬는 2차원배렬로 표시된다고 가정하시오. 다음과 같은것을 수행하는 프로그람을 작성하시오.

ㄱ. 그루빠가 주어 졌을 때 그 그루빠의 크기를 계산하시오.

ㄴ. 각이한 그루빠들의 수를 계산한다.

ㄷ. 모든 그루빠들을 목록으로 표시하시오.

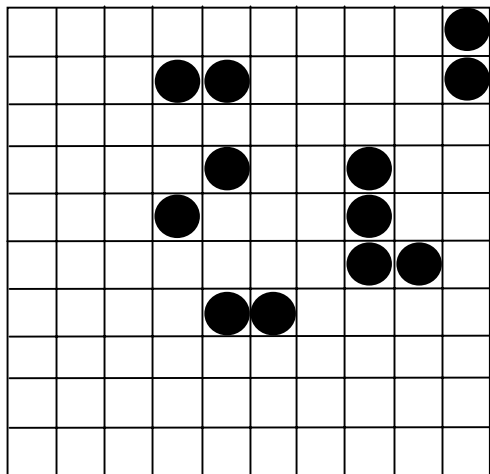



그림 9-55. 련습문제 9-45의 격자칸

9-46. 제8장 제7절에서 미로생성을 서술하였다. 미로의 경로를 출력하려고 한다고 하자. 미로는 행렬로 표시된다고 가정하시오. 이때 행렬의 매 세포는 어느 벽들이 존재하는가 존재하지 않는가에 대한 정보를 보관한다.

ㄱ. 미로의 경로를 출력하는데 충분한 정보를 계산하는 프로그람을 작성하시오. 결과 SEN…형태로 주시오(남쪽으로 가서 그다음은 동쪽, 다음은 왼쪽 등을 표시하는).

ㄴ. 윈도우프로그람을 가진 체계(Visual C++와 같은)를 리용하여 미로를 표시하고 단추를 눌러서 경로를 그리는 프로그람을 작성하시오.

9-47. 미로의 벽들이 P개의 4각형들의 추가로 해체될수 있다고 가정하시오. P는 알고리듬에서 파라메터로 규정된다(추가되는것이 0이라면 그때 이 문제는 자명하다.). 이 문제의 어떤 갱신판을 작성하는 알고리듬을 서술하시오. 그 알고리듬의 실행시간은 얼마인가?

9-48. 미로가 풀이를 가지지 않을수도 있다고 가정하시오.

ㄱ. 풀이를 얻기 위해서 해체되여야 할 벽의 최소수를 결정하는 선형시간알고리듬을 작성하시오(암시: 쌍방향대기렬을 사용하시오.).

ㄴ. 최소수의 벽을 해체한 후의 최단경로를 찾아 내는 알고리듬(반드시 선형시간은 아닌)을 작성하시오. ㄱ의 풀이는 어느 벽들이 해체되는것이 가장 좋은가하는 정보를 주지 않는다는것에 주의하시오(암시:련습문제 9-47을 리용하시오.). 련습문제 9-49로부터 9-53까지의 매 문제에 최단경로알고리듬을 적용하여 어떻게 풀수 있는가를 설명하시오. 그다음 입력값을 표시하는 체계를 설계하고 그 문제를 푸는 프로그람을 작성하시오.

9-49. 입력값은 련맹전점수목록(여기에서 무승부는 없다.)이다. 만일 모든 팀들이 적어도 한번은 이기고 진다면 일반적으로 어느 팀이 다른 어떤 팀보다 더 잘한다는것을 보잘것없는 이행성 방법으로 《증명》할수 있다. 실례로 매개 팀이 3번 경기를 하는 6팀련맹전에서 다음의 결과가 나왔다고 하자. A는 B와 C를 이기고 B는 C와 F를 이기고, C는 D를 이기고, D는 E를 이기고 E는 A를 이기고 F는 D와 E를 이겼다. 그때 A는 반대로 F를 이긴 B를 이겼기때문에 A는 F보다 더 좋다고 증명할수 있다. 이와 류사하게 F가 E를 이기고 E가 A를 이겼기때문에 F는 A보다 더 좋다고 증명할수 있다. 경기성적목록과 두 팀 X, Y가 주어 졌을 때 X가 Y보다 더 좋다는 증거(만일 있다면)를 찾으시오. 또는 이러한 형식의 그 어떤 증거를 찾을수 없다는것을 지적하시오.

9-50. 하나의 단어는 한개 문자의 치환으로 다른 단어로 변화될수 있다. 5문자단어들이 있는 사전이 있다고 가정하시오. 단어 A가 한개문자치환의 서렬들에 의

해 단어 B로 변화될수 있는가를 결정하고 만일 된다면 해당한 단어들의 서렬을 출력하는 알고리듬을 작성하시오. 모든 중간단어들은 사전에 있어야 한다. 실례로 bleed는 렬 bleed blend, blond, blood에 의해 blood로 바뀐다.

9-51. 입력값은 화폐들과 그것들의 교환률이다. 즉석에서 돈을 처리하는 교환들의 서렬이 있는가? 실례로 화폐들로는 X, Y, Z가 있고 교환률은 1X는 2Y와 같고 1Y는 2Z와 같으며 1X는 3Z와 같을 때 300Z로는 100X를 사고 이것은 다시 200Y를 사며 다시 반대로 400Z를 살수 있다. 이렇게 하여 33%의 리익을 얻는다.

9-52. 학생은 졸업할 과정안의 수가 몇개인가를 정확히 알 필요가 있으며 매 과정안들은 집행되여야 할 필수과목들을 가지고 있다. 모든 과정안들은 매 학기 선정되며 학생은 과정안들을 제한없이 가질수 있다고 하자. 과정안들의 목록과 그 필수과목들이 주어 졌을 때 학기의 최소수를 요구하는 일람표를 계산하시오.

9-53. Kevin Bacon Game의 목적은 영화배우를 공유된 배역들을 통하여 Kevin Bacon에 련결하는것이다. 련결들의 최소수는 하나의 배우의 Bacon수이다. 실례로 Tom Hanks는 Bacon를 1을 가지고 있는데 그는 Kevin Bacon과 함께 Apollo13내에 있었다. Sally Fields는 그의 Bacon수 2를 가지고 있는데 그 녀자는 Tom Hanks와 Forest Gump에 있었다. Tom Hanks는 Kevin Bacon과 Hppollo 1 3에 있었다. 대부분의 모두 잘 알려 진 배우들은 Bacon number 1이나 2를 가진다. 당신에게 배역을 맡은 배우들의 포괄적인 목록이 있다고 가정하고 다음과 같은 것을 하시오.

ㄱ. 배우의 Bacon번호를 발견하는 방법을 설명하시오.

ㄴ. 가장 높은 Bacon번호를 가진 배우를 발견하는 방법을 설명하시오.

ㄷ. 임의의 두명의 배우들사이의 련계의 최소수를 발견하는 방법을 설명하시오.

9-54. clique문제는 다음과 같이 규정할수 있다. 즉 무방향그라프 $G=(V,E)$와 옹근수 K가 주어 지면 G는 적어도 K개의 정점들로 된 완전부분그라프를 포함하는가? ≪정점포괄≫문제는 다음과 같이 규정할수 있다. 무방향그라프 $G=(V,E)$와 옹근수 K가 주어 지면 G는 $|V'|\leq K$이고 G의 매개 변이 V'에서 하나의 정점을 가지는 부분모임 $V'\subset V$를 포함하는가? clique문제는 정점포괄에 다항식적으로 변환가능하다는것을 보여 주시오.

9-55. 무방향그라프들에서 하밀톤순환경로문제가 NP-완전하다고 가정하자.

ㄱ. 하밀톤순환경로문제는 방향그라프들에서 NP완전성이라는것을 증명하시오.

ㄴ. 무게없는단순최대경로문제는 방향그라프들에서 *NP*-완전성이라는것을 증명하시오.

9-56. 놀이카드수집문제는 다음과 같다. P_1, P_2, ⋯, P_M이라는 조가 주어 지고 매 조가 2개의 야구카드의 부분모임과 옹근수 *K*를 포함한다면 *K*번이상 조들을 선택하여 모든 야구카드를 수집할수 있겠는가? 야구카드수집문제는 *NP*-완전성이라는것을 증명하시오.

참고문헌

그라프리론에 대한 좋은 책들은 [8], [13], [22], [37]들이다. 실행시간에 대하여 더 구체적으로 서술한 책들은 [39], [41], [48]들이다.

린접목록들의 리용은 [24]에 제시되였다. 위상학적정렬알고리듬은 [29]에 있으며 [34]에도 서술되였다. 딕스트라알고리듬은 [9]에 서술되였다. d-더미들과 피보나치더미들에 대한 개선된 내용들은 각각 [28]과 [15]에 서술되여 있다. 부무게값을 가지는 최단경로알고리듬은 밸맨의 [3]에 있으며 타잔의 [48]은 그 완성에 대한 더 효과적인 방법을 서술하였다.

망흐름문제에 대한 포도와 플카손의 유력한 론리는 [14]에 있다. 최단경로에 대한 문제나 최대흐름을 허용하는 경로들에 대한 문제는 [14]에 있다. 이 문제에 대한 다른 론문들은 [10], [32], [21], [6], [32]에서 참고할수 있다. 최소비용흐름문제에 대한 알고리듬은 [19]에서 참고할수 있다. 초기의 최소생성나무알고리듬은 [4]에서 참고할수 있다. 프림알고리듬은 [42]에, 크루스칼알고리듬은 [35]에 있다. 두개의 $O(|E|\log\log|V|)$알고리듬들은 [5]와 [49]에 있다. 리론적으로 가장 잘 알려진 알고리듬들은 [15], [17], [30]에 있다. 이 알고리듬에 대한 경험적인 연구는 decreasekey로 실현되는 프림알고리듬이 대부분의 그라프들에서 실천적으로 가장 좋다는것을 보여 준다.([40])

쌍련결성에 대한 알고리듬은 [44]에 있다. 초기의 강한 련결성분을 구하는 선형시간알고리듬(련습문제 4-28)은 같은 책에 있다. 이 책에서 표현된 그에 대한 알고리듬은 크사라크(비공개된)와 샤리[43]에 있다. 깊이우선탐색에 대한 다른 응용들은 [25], [26], [45], [46]에 있다(제8장에서 서술한것처럼 [45]와 [46]의 결과들은 개선되였지만 기본알고리듬은 변화되지 않았다.).

NP-완전성문제들에 대한 리론적인 내용들은 [20]에 있다. 추가적인 내용들은 [1]에서 참고할수 있다. 만족성에 대한 NP-완전성문제들은 [7]에서 보여 진다. 이에 대한 또 하나의 유명한 론문은 [31]인데 여기에서는 NP-완전성에 대한 21개의 문제들이 서술되였다. 복잡성리론에 대한 우수한 론문은 [47]이다. 순회판매원문제에 대한 근사알고리듬은 일반적으로 아주 최적인 결과를 주는데 그것은 [38]에서 참고할수 있다.

련습문제 9-8에 대한 풀이는 [2]에서 찾아 볼수 있다. 련습문제 9-13에 있는 2진정합문제에 대한 풀이는 [23]과 [36]에서 찾아 볼수 있다. 이 문제는 변들에 무게를 추가하고 그 그라프를 둘로 나누는 제한조건들을 제거하여 만들수 있다. 일반그라프들에서 무게없는 결합문제에 대한 효과적인 풀이는 아주 복잡하다. 그 상세한 내용들은 [11]과 [16], [18]에서 참고할수 있다.

련습문제 9-35는 실천에서 일반적으로 리용되는 평면그라프를 취급하였다. 평면그라프들은 대단히 성글며 많은 어려운 문제들은 평면그라프들에서 더 쉽다. 하나의 실례는 그라프동형문제인데 이것은 평면그라프문제들을 선형시간에 풀수 있다[27]. 일반그라프들에서 다차원시간알고리듬은 알려 진것이 없다.

1. A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms,* Addison-Wesley, Reading, Mass., 1974.
2. R. K. AhuJa, K. Melhorn, J. B. Orlin, and R. E. Tarjan, "Faster Algorithms for the Shortest Path Problem," *Journal of the ACM,* 37 (1990), 213-223.
3. R. R. Bellman, "On a Routing Problem," *Quarterly of Applied Mathematics,* 16 (1958), 87-90.
4. Boruvka, "Ojistem problemu minimalnim (On a Minimal Problem)," *Praca Moravske Prirodovedecke Spolecnosti,* 3 (1926), 37-58.
5. D. Cheriton and R. E. Tarjan, "Finding Minimum Spanning Trees," *SIAM Journal on Computing, 5* (1976), 724-742.
6. J. Cheriyan and T. Hagerup, "A Randomized Maximum-Flow Algorithm," *SIAM Journal ournal on Computing,* 24 (1995), 203-226.
7. S. Cook, "The Complexity of Theorem Proving Procedures," *Proceedings of the Third Annual ACM Symposium on Theory of Computing* (1971), 151-158.
8. N. Deo, *Graph Theory with Applications to Engineering and Computer Science*, Prentice Hall, Englewood Cliffs, NJ., 1974. .
9. E. W. Dijkstra, "A Note on Two Problems in Connexion with Graphs," *NumerischeMathematik,* 1 (1959), 269-271.
10. E. A. Dinic, "Algorithm for Solution of a Problem of Maximum Flow in Networks with Power Estimation," *Soviet Mathematics Doklady,* 11 (1970), 1277-1280.
11. J. Edmonds, "Paths, Trees, and Flowers," *Canadian Journal of Mathematics,* 17 (1965) 449-467.
12. J. Edmonds and R. M. Karp, "Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems," *Journal of the ACM,* 19 (1972), 248-264.
13. S. Even, *Graph Algorithms,* Computer Science Press, Poromac, Md., 1979.
14. L. R. Ford, Jr., and D. R. Fulkerson, *Flows in Networks,* Princeton University Press, Princeton, N.J., 1962.

15. M. L. Fredman and R. E. Tarjan, "Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms," *Journal of the A*CM, 34 (1987), 596-615.
16. H. N. Gabow, "Data Structures for Weighted Matching and Nearest Common Ancestors with Linking," *Proceedings of First Annual ACM.-SIAM Symposium on Discrete Algorithms* (1990), 434-443.
17. H. N. Gabow, Z. Galil, T. H. Spencer, and R. E. Tarjan, "Efficient Algorithms for Finding Minimum Spanning Trees on Directed and Undirected Graphs," *Comhinatorica*, 6(1986), 109-122.
18. Z. Galil, "Efficient Algorithms for Finding Maximum Matchings in Graphs," *ACM Computing Surveys*, 18 (1986), 23-38.
19. Z. Galil and E. Tardos, "An $O(n^2(m+nlogn)logn)$ Min-Cost Flow Algorithm," *Journal of the ACM, 35* (1988), 374-386.
20. M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP'-Completeness*, Freeman, San Francisco, 1979.
21. A. V. Goldberg and R. E. Tarjan, "A New Approach to the Maximum-Flow Problem," *J ournal of the ACM, 35* (1988), 921-940.
22. F. Harary, *Graph Theory,* Addison-Wesley, Reading, Mass., 1969.
23. J. E. Hopcroft and R. M. Karp, "An $n^{5/2}$ Algorithm for Maximum Matchings in Bipartite Graphs," *S I AM Journal on Computing,* 2 (1973), 225-231.
24. J. E. Hopcrofr and R. E. Tarjan, "Algorithm 447: Efficient Algorithms for Graph Manipulation," *Communications of the* ACM, 16 (1973), 372-378.
25. J. E. Hopcroft and R. E. Tarjan, "Dividing a Graph into Triconnected Components," *SIAM Journal ournal on Computing,* 2 (1973), 135-158.
26. J. E. Hopcroft and R. E. Tarjan, "Efficient Planarity Testing," *Journal of the* ACM, 21 (1974),549-568.
27. J. E. Hopcroft and J. K. Wong, "Linear Time Algorithm for Isomorphism of Planar Graphs," *Proceedings of the Sixth Annual* ACM *Symposium on Theory of Computing* (1974), 172-184.
28. D. B. Johnson, "Efficient Algorithms for Shortest Paths in Sparse Networks," *Journal of the ACM,* 24 (1977), 1-13.
29. A.B. Kahn, "Topological Sorting of Large Networks," *Communications of the* ACM, 5 (1962), 558-562.
30. D. R. Karger, P. N. K l ein, and R. E. Tarjan, "A Randomized Linear-Time Algorithm to Find Minimum Spanning Trees," *Journal of the ACM,* 42 (1995), 321-328.
31. R. M. Karp, "Reducibility among Combinatorial Problems," *Complexity of Computer Computations* (eds. R. E. Miller and J. W. Thatcher), Plenum Press, New York, 1972, 85-103.
32. A.V. Karzanov, "Determining the Maximal Flow in a Network by the Method of Preflows，” *Soviet Mathematics Doklady,* 15 (1974), 434-437.

33. V. King, S. Rao, and R. E. Tarjan, "A Faster Deterministic Maximum Flow Algorithm," *Journal of Algorithms,* 17 (1994), 447-474.
34. D. E. Knuth, *The Art of Computer Programming, Vol. 1: Fundamental Algorithms,* 3d. ed., Addison-Wesley, Reading, Mass., 1997.
35. J. B. Kruskal, Jr., "On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem," *Proceedings of the American Mathematical Society, 7* (1956), 48-50.
36. H. W. Kuhn, "The Hungarian Method for the Assignment Problem," *Naval Research Logistics Quarterly,* 2 (1955), 83-97.
37. E. L. Lawler, *Combinatorial Optimization: Networks and Matroids,* Holt, Reinhart, and Winston, New York, 1976.
38. S. Lin and B. W. Kernighan, "An Effective Heuristic Algorithm for the Traveling Salesman Problem," *Operations Research,* 21 (1973), 498-516.
39. K. Melhorn, *Data Structures and Algorithms 2: Graph Algorithms and NP-completeness,* Springer-Verlag, Berlin, 1984.
40. B. M. E. Moretand H. D. Shapiro, "An Empirical Analysis of Algorithms for Constructing a Minimum Spanning Tree," *Proceedings of the Second Workshop on Algorithms and Data Structures* (1991), 400-411.
41. C. H. Papadimitriou and K. Steiglitz, *Combinatorial Optimization: Algorithms and Complexity,* Prentice Hall, Englewood Cliffs, N.J., 1982.
42. R. C. Prim, "Shortest Connection Networks and Some Generalizations," *Bell System Technical Journal,* 36 (1957), 1389-1401.
43. M. Sharir, "A Strong-Connectivity Algorithm and Its Application in Data Flow Analysis," *Computers and Mathematics with Applications, 7* (1981), 67-72.
44. R. E. Tarjan, "Depth First Search and Linear Graph Algorithms," *SIAM Journal on Computing, 1* (1972), 146-160.
45. R. E. Tarjan, "Testing Flow Graph Reducibility," *Journal of Computer and System Sciences, 9* (1974), 355-365.
46. R. E. Tarjan, "Finding Dominators in Directed Graphs," *SIAM Journal on Computing, 3* (1974), 62-89.
47. R. E. Tarjan, "Complexity of Combinatorial Algorithms," *SIAM Review,* 20 (1978), 457-491.
48. R. E. Tarjan, *Data Structures and Network Algorithms,* Society for Industrial and Applied Mathematics, Philadelphia, 1983.
49. A. C. Yao, "An 0(｜E｜ 1 og log ｜V｜) Algorithm for Finding Minimum Spanning Trees," *Information Processing Letters,* 4 (1975), 21-23.

제10장. 알고리듬설계기술

지금까지는 알고리듬의 효과적인 실현에 주의를 돌려 왔다. 알고리듬이 주어 졌을 때 실제적인 자료구조는 설명할 필요가 없다고 보았다. 실행시간이 될수록 적게 걸리도록 적당한 자료구조를 선택하는것은 프로그람작성자에게 달려 있다.

이 장에서는 알고리듬의 실현으로부터 알고리듬의 설계에로 화제를 바꾼다. 지금까지 본 대부분의 알고리듬은 직선적이고 단순하다. 제9장에는 미묘한 처리를 요구하는 알고리듬들이 있다. 일부 알고리듬은 실지 정확하다는것을 보여 주기 위하여 증명을 요구한다. 이 장에서 실지 문제를 해결하는데 사용되는 알고리듬의 일반형태들중 5개에 주목을 돌린다. 많은 문제들에서 이 방법들은 아주 정확히 수행된다. 특히 알고리듬의 매 형태들에 대해

- 일반적인 고찰을 주고
- 여러 실례들을 들어 보며(장의 마지막에 있는 련습문제들은 매우 많은 실례들을 제공한다.)
- 일반적인 항목들로 근사적인 시간복잡도와 공간복잡도들을 설명한다.

제1절. 탐욕알고리듬

시험해 보려는 알고리듬의 첫 형태는 《**탐욕알고리듬**》이다. 이미 제9장에서 딕스트라, 프림, 크루스칼의 알고리듬들 즉 3개의 탐욕알고리듬들을 보았다. 탐욕알고리듬들은 여러 단계로 수행된다. 다음 단계의 결과에 무관계하게 매 단계에서 되도록 괜찮아 보이는 결심을 하여야 한다. 일반적으로 이것은 일부 《어떤 국부적인 최적처리》를 선택한다는것을 의미한다. 《현재 손에 넣을수 있는것을 선택하라.》는 내용을 반영한 이 방법으로부터 이러한 부류의 알고리듬들의 이름이 유래되였다. 알고리듬을 완성할 때 알고리듬의 국부적인 최적처리가 《전체적인 최적처리》와 같아 지기를 바란다. 만일 국부적인 최적처리가 전체 최적처리와 같다면 그 알고리듬은 정확하며 만약 그렇지 않으면 그 알고리듬은 부분최적인 풀이를 얻어 낸다. 절대적으로 가장 정확한 대답이 요구되지 않으면 일반적으로 정확한 결과를 얻는데 쓰는 더 복잡한 알고리듬들을 리용하기보다는 차라리 단순한 탐욕알고리듬들을 리용하여 근사적인 결과를 얻는것이 낫다.

실생활적인 탐욕알고리듬들에 대한 몇가지 실례들이 있다. 가장 대표적인것은 **동전교환알고리듬**이다. 미국통화로 잔돈을 바꾸기 위해 반복적으로 가장 큰 단위화페를 처리한다. 17딸라와 61쎈트를 잔돈으로 나누어 주기 위하여 10딸라지페 1장, 5딸라지페 1장,

한딸리짜리 지페 2장, 25쎈트짜리 잔돈 2개, 10쎈트짜리잔돈 1개 그리고 1페니를 나누어 준다. 이렇게 함으로써 지폐와 쇠돈들의 수를 최소화한다. 이 알고리듬은 모든 통화체계들에서 리용할수 없고 미국통화체계에서만 리용할수 있다. 실지로 이 알고리듬은 2딸라짜리 지폐와 50쎈트짜리 잔돈들의 경우에도 적용된다.

국부적인 최적선택이 언제나 가능한것이 아닌 하나의 실례로써 교통문제를 들수 있다. 실례로 미아미에서 퇴근시 혼잡한 시간동안에는 교통이 도로상에서 한마일씩이나 정지되고 사람들이 오도가도 못하게 되기때문에 거리들이 텅 비여 보인다 하더라도 기본거리는 피하는것이 좋다. 더 마비되면 모든 교통장애를 피하기 위하여 일부 경우에는 목적지와는 반대방향으로 림시 우회하는것이 더 좋다.

이 절의 마지막부분에서는 탐욕알고리듬들을 리용하는 몇가지 응용프로그람들을 고찰한다. 첫번째 응용프로그람은 간단한 일정작성문제이다. 가상적으로 모든 일정작성문제들은 *NP*완전성(혹은 류사하게 어려운 복잡성이 있는)이거나 혹은 탐욕알고리듬으로 풀수 있다. 두번째 응용프로그람은 파일을 압축하는 문제인데 이것은 콤퓨터과학의 가장 최신성과들중의 하나이다. 최종적으로 근사적인 탐욕알고리듬에 대한 실례를 고찰한다.

1. 간단한 일정작성문제

실행시간이 각각 $t_1, t_2, \cdots, t_N$인 일감 $j_1, j_2, \cdots, j_N$이 주어 졌다고 하자. 하나의 단순처리기를 가지고 있을 때 평균완성시간이 최소가 되도록 이 일감들의 순서를 작성하기 위한 가장 좋은 방도는 무엇인가? 이 절에서는 비선취권일정작성을 가정하는바 한 일감이 시작되면 그것은 완성될 때까지 진행되여야 한다.

실례로 표 10-1에서와 같이 4개의 일감들과 그의 실행시간들이 있다고 하자. 한가지 가능한 일정을 그림 10-1에 보여 주었다. j_1은 15(시간단위)만에, j_2는 23, j_3은 26, j_4는 36만에 끝나기때문에 평균완성시간은 25이다.

표 10-1. 일감과 시간

일감	시간
j_1	15
j_2	8
j_3	3
j_4	10

표 10-2. 일감과 시간

일감	시간
j_1	3
j_2	5
j_3	6
j_4	10
j_5	11
j_6	14
j_7	15
j_8	18
j_9	20

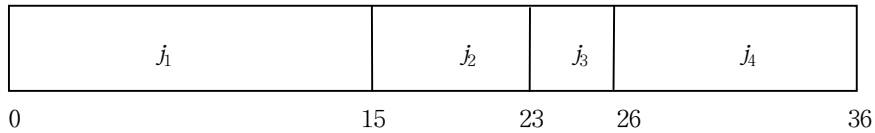



그림 10-1. 일정작성 #1

더 좋은 일정은 17.75의 완성시간을 주는데 그것은 그림 10-2에 보여 주었다.

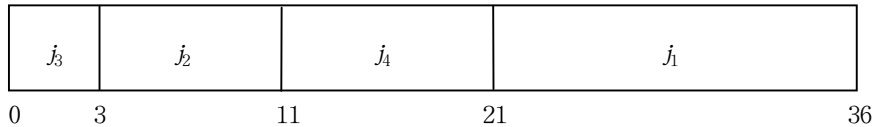



그림 10-2. 일정작성 #2(최적일정)

그림 10-2에 보여 준 일정에서 일감들은 가장 짧은 시간순서로 정돈된다. 이것은 언제나 최적일정을 산출한다는것을 보여 줄수 있다. 일정의 일감들을 $j_{i_1}, j_{i_2}, \cdots, j_{i_N}$ 이라고 하자. 첫 일감은 t_{i_1} 시간내에 끝난다. 두번째 일감은 $t_{i_1}+t_{i_2}$ 시간후에 끝난다. 그리고 세번째 일감은 $t_{i_1}+t_{i_2}+t_{i_3}$ 시간내에 끝난다. 이로부터 공정의 총 원가 C는

$$C = \sum_{k=1}^{N}(N-k+1)t_{i_k} \tag{10-1}$$

$$C = (N+1)\sum_{k=1}^{N}t_{i_k} - \sum_{k=1}^{N}k \cdot t_{i_k} \tag{10-2}$$

이다.

식 10-2에서 첫 합은 일감의 순서작성에 무한계하며 오직 두번째 합만이 전체 원가에 영향을 미친다. 순서작성에서 $t_{i_x} < t_{i_y}$ 인 어떤 $x > y$가 존재한다고 하자. 이때 j_{i_x} 와 j_{i_y} 를 서로 바꾸어 계산함으로써 두번째 합을 증가시키고 전체 원가를 줄일수 있다. 이렇게 하여 시간이 단조비감소하지 않는 일감들의 임의의 일정을 부분최적화할수 있다. 남은 유일한 일정들은 처음에 최소실행시간에 따라 정돈되는 일감들인데 임의로 련계를 끊어 버린다.

이 결과는 조작체계의 일감처리기가 일반적으로 더 짧은 일감들에 우선권을 주게 되는 리유를 지적한다.

다중처리기의 경우

우의 문제를 처리기가 여러개인 경우에로 확장할수 있다.

P개의 처리기들과 실행시간이 각각 $t_1, t_2, \cdots, t_N$을 가지는 일감 $j_1, j_2, \cdots, j_N$이 있다고 하자. 처음에 일감들은 가장 짧은 실행시간에 따라 정돈되다고 하자. 실례로 일감이 P=3일 때 표 10-2와 같다고 하자.

그림 10-3에서는 평균완성시간을 최소화하는 한가지 최적정돈상태를 보여 주었다. 일감 j_1, j_4, j_7은 처리기 1에서 실행된다. 처리기 2는 j_2, j_5, j_8을 조종한다. 그리고 처리기 3은 나머지일감들을 실행한다. 일감을 완성하는데 걸리는 전체 시간은 165인데 그 평균값은 165/9로서 18.33이다.

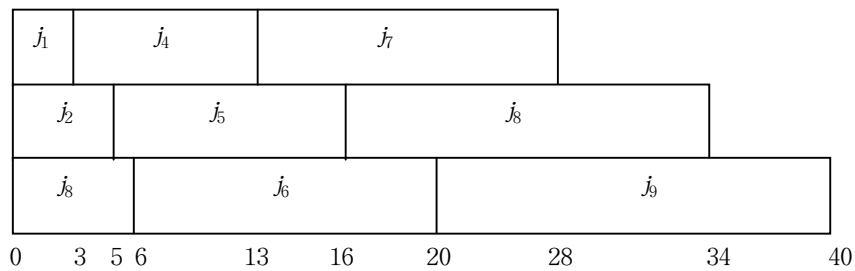



그림 10-3. 다중처리기인 경우 한가지 최적풀이

다중처리기의 경우를 해결하는 이 알고리듬에서는 처리기들을 따라 순환하면서 일감들을 순서대로 시작한다. 더 좋은 다른 순서화가 있는가 없는가를 어렵지 않게 볼수 있다. 처리기수 P가 일감수 N을 나머지없이 나눈다면 수많은 최적순서화가 있게 된다. 이것은 매개 $0 \leq i < N/p$에 대하여 j_{1P+1}에서부터 $j_{(i+1)P}$까지의 매개 일감들을 각각 매 처리기에 할당하여 얻을수 있다. 그림 10-4는 이 경우의 두번째 최적풀이방안을 보여 주었다.

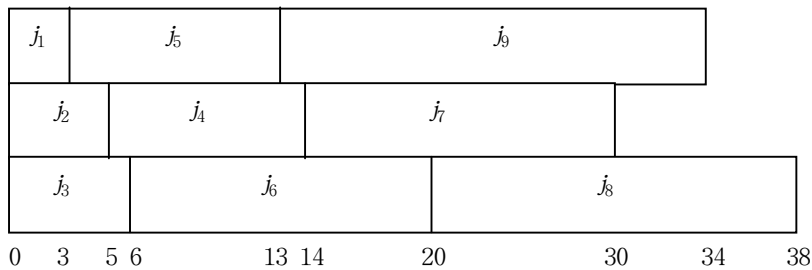



그림 10-4. 다중처리기인 경우 두번째 최적풀이

가령 P가 N 으로 완전히 나누어 떨어 지지 않고 모든 일감시간들이 각이하다고 해도 여전히 많은 최적해결방안들이 있을수 있다. 이것을 련습문제에서 풀 과제로 남겨 둔다.

최종완성시간의 최소화

아주 간단한 문제를 고찰하는것으로써 이 부분을 끝내려 한다. 마지막일감이 끝나는 시점에 대해서만 고찰해 보자. 우에서 제시한 두개의 실례에서 이 완성시간은 각각 40과 38이다. 그림 10-5에서는 마지막완성시간이 제일 적은것이 34이며 이것은 모든 처리기가 항상 가동하기때문에 더이상 개선될수 없다는것을 명백하게 보여 주었다.

이 계획은 평균완성시간을 최소화하지는 못하더라도 전체 렬의 완성시간을 더 짧게 하는데 기여하게 된다는것을 보여 준다. 만일 같은 사용자들이 이런 일감들을 가진다면 그때 이것은 완전한 일정작성방법으로 된다. 이 문제들은 아주 간단하다고 하여도 이 새로운 문제는 *NP*-완전성문제로 처리하는데 이것은 이 절의 뒤에서 보게 될 상자채우기문제나 배낭채우기문제를 푸는 또 하나의 방법으로 된다. 그러므로 최종완성시간을 최소화하는것은 명백히 중간평균완성시간을 최소화하는것보다는 더 어렵다.

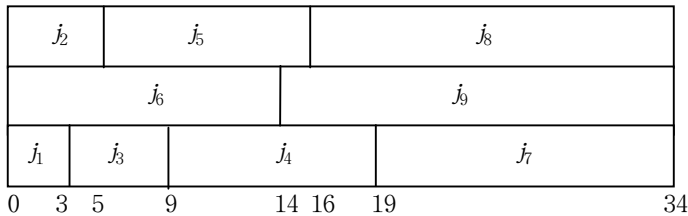



그림 10-5. 최종완성시간의 최소화과정

2. 하프만부호

이 절에서는 **파일압축**(*File compression*)이라고 하는 탐욕알고리듬의 두번째 응용을 고찰한다.

일반적인 아스키문자모임은 대체로 100여개의 《출력가능한》문자들로 이루어 진다. 이 문자들을 구별하는데 [log100]=7bit의 길이가 요구된다. 7개의 비트로서는 128개의 문자를 표시할수 있으며 아스키문자모임에 여러개의 《출력불가능한》문자들을 더 추가할수 있다. 여덟번째 비트는 기우성검사비트로 리용된다. 중요한것은 문자모임의 크기가 C라면 [log*C*]개의 비트들이 표준부호화에 리용된다는것이다.

문자 *a, e, i, s, t*에다 공백문자와 행바꾸기부호까지만을 포함하는 파일이 있다고 하자. 이 파일은 10개의 *a*와 15개의 *e*, 12개의 *i*, 3개의 *s*, 4개의 *t*, 13개의 공백과 하나의 행바꾸기부호들로 되여 있다고 하자. 표 10-3에서 보여 준것처럼 이 파일을 표시하는데 총 58개 문자가 필요하고 매 문자당 3bit가 필요하므로 174개의 비트가 요구된다.

실제로는 파일들이 매우 클수 있다. 거의 대다수 파일들은 어떤 프로그람의 결과자료들이며 가장 출현빈도가 높은 문자와 가장 출현빈도가 낮은 문자사이에는 보통 큰 차

이가 생긴다. 례를 들면 대다수의 방대한 파일들은 지나치게 큰 수자들과 공백들, 행바꾸기들을 가지지만 문자 q와 x의 빈도수는 비교적 작다. 여기서 흥미를 끄는것은 그 자료들을 상대적으로 속도가 뜬 매체를 통하여 전송하려고 할 때 파일의 크기를 감소시키는 문제이다. 사실 매 콤퓨터에서 디스크의 공간은 매우 귀하므로 사람들은 질 좋은 부호를 작성하면서도 요구되는 총 비트수를 줄일수 없겠는가 하고 생각하게 된다.

표 10-3. 표준부호화체계의 리용

문 자	부 호	반복수	전체 비트수
a	000	10	30
e	001	15	45
i	010	12	36
s	011	3	9
t	100	4	12
공백	101	13	39
행바꾸기	110	1	3
합계		58	174

결과는 이것이 가능하며 간단한 방법으로 일반적으로 큰 파일에 대해서는 25%, 대단히 큰 파일들에 대해서는 50~60%만큼 기억량을 줄일수 있다는것이다. 일반적인 방법은 부호길이를 문자로부터 문자로 변화시키는것이며 자주 출현하는 문자들은 짧은 부호길이를 가지도록 하는것이다. 사실 모든 문자들이 같은 빈도로 출현하면 그 어떤 절약에 대하여 생각할수 없다.

자모를 표시하는 2진부호는 그림 10-6에서 보여 준 **2진나무**(*bynary tree*)로 표시할수 있다.

그림 10-6에서 보여 준 나무는 오직 잎에만 자료를 가지고 있다. 매 문자는 뿌리에서 출발하여 경로를 기록하면서 표시할수 있는데 왼쪽 가지를 지적하는데 0을, 오른쪽 가지를 지적하는데 1을 리용한다. 례를 들면 s는 왼쪽 가지로 옮겨 갔다가 다시 오른쪽 가지로 가며 마지막으로 다시 오른쪽으로 옮겨 간다. 이것은 011로 부호화된다. 이 자료구조는 때때로 **트라이나무**(*trie*)에 귀착된다. 만일 문자 c_i가 그의 길이 d_i로써 f_i번 출현한다면 부호계산량은 $\sum d_i f_i$ 이다. 그림 10-6에 보여 준것보다 더 좋은 부호는 행바꾸기가 유일한 자식이라는것을 리용하여 얻을수 있다. 행바꾸기기호를 한준위 더 높여 그의 부모에 배치함으로써 그림 10-7에 보여 준 새로운 나무를 얻을수 있다. 이 새로운 나무의 계산량은 173이지만 최적량보다는 아직 멀다.

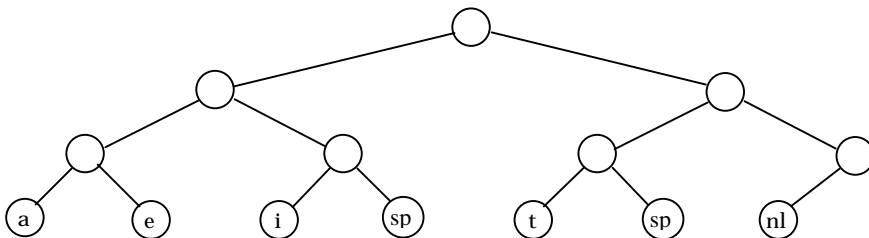



그림 10-6. 나무에서 초기부호의 표현

그림 10-7에 보여 준 나무는 완전나무이다. 즉 모든 매듭들은 모두 잎이거나 또는 두개의 자식들을 가지고 있다. 최적인 부호는 늘 이런 특성이 있으며 그렇지 않으면 이미 보아 온것처럼 하나의 자식만 가지고 있을 때 웃준위에로 올라 갈수 있다.

이런 인자들을 다시 보면 기본문제는 총 계산량이 최소인 완전2진나무를 구하는것인데 여기에서도 모든 문자들은 잎들에 포함된다. 그림 10-8에 보여 준 나무는 우의 실례에 대한 문자들의 최적나무를 보여 준다. 표 10-4에서 볼수 있는바와 같이 이 부호는 146개의 비트만을 리용한다.

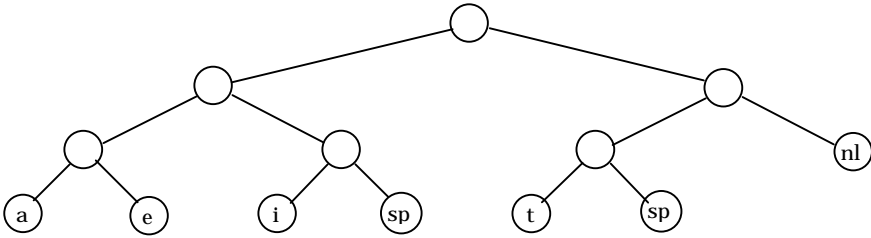



그림 10-7. 좀 더 개선된 나무

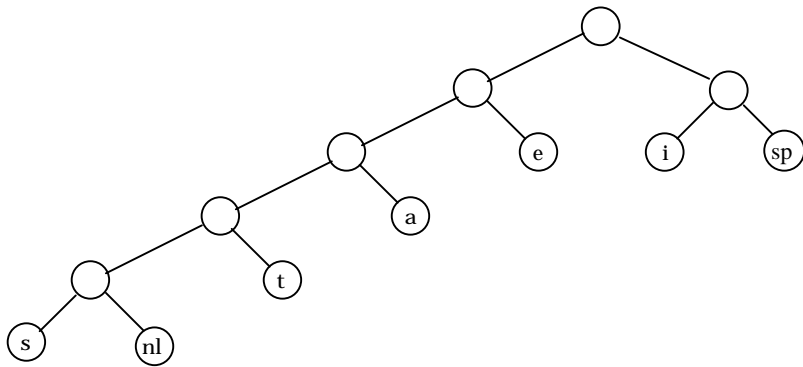



그림 10-8. 최적화된 **앞배치(prefix)부호**

표 10-4. 최적화된 prefix부호

문자	부호	반복수	합비트수
a	001	10	30
e	01	15	30
i	10	12	24
S	00000	3	15
T	0001	4	16
공백	11	13	26
행바꾸기	00001	1	5
총 비트수			146

이 나무에는 최적부호가 많다. 이것들은 부호화나무에서 자식들을 교체하여 얻을수 있다. 아직 결정되지 않은 기본문제는 이제 어떻게 부호화나무를 만드는가 하는것이다. 이것을 실현할수 있는 알고리듬은 1952년에 하프만이 제기했다. 이러한 부호화체계를 일

부호(*Huffman code*)라고 한다.

하프만알고리듬

이 절에서 쓴 총 문자들의 수를 C라고 한다. 하프만알고리듬은 다음과 같다. 나무들의 수림을 보관한다. 어떤 나무의 **무게**(*weight*)는 그 잎들의 출현빈도수와 같다. 가장 작은 무게를 가지는 나무를 선택하여 T_1과 T_2를 C-1번 그것도 무게가 가장 작은것으로 결합하여 새 나무(보조나무를 가진)를 만든다. 알고리듬의 시작점에는 C개의 단일매듭나무가 있는데 매개 기호는 하나의 나무에 대응된다. 알고리듬의 마지막에는 하나의 나무가 있는데 이것이 최적인 하프만부호나무이다.

하나의 실례로 알고리듬의 처리과정을 명백히 고찰하자. 그림 10-9는 초기의 수림을 보여 주는데 매 나무의 무게는 그 뿌리에 있는 작은 수자로 표시된다. 무게가 가장 작은 두개의 나무를 합쳐서 그림 10-10에서 보여 준 수림을 형성한다. 그 새로운 뿌리를 *T*1로 하며 이렇게 하여 앞으로의 합치기가 명백하게 시작된다. *s*를 왼쪽 자식으로 하여 임의의 결합이 진행될수 있다. 새 나무의 총 무게는 바로 그전 나무들의 무게의 합으로 쉽게 계산될수 있다. 새 나무를 만드는것 역시 간단한데 간단히 새 매듭을 얻어야 할 필요가 있으면 왼쪽과 오른쪽 점을 설정하고 무게를 기록한다.

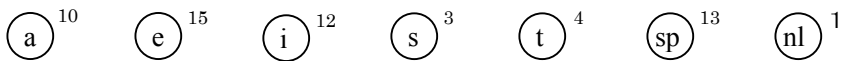



그림 10-9. 하프만알고리듬의 초기 수림

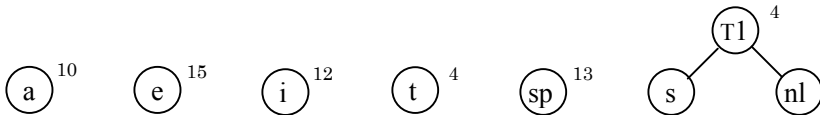



그림 10-10. 첫 결합을 한후의 하프만알고리듬

이제 여섯개의 나무가 있는데서 다시 가장 무게가 작은 두개의 나무를 선택한다. 이렇게 세번째 결합이 진행된후 가장 무게가 작은 두 나무는 바로 단일매듭나무로 표시된 *i*와 공백기호이다. 하여 *T*1과 *t*가 생기는데 뿌리가 *T*2이고 무게가 8인 새로운 나무와 합친다. 이것을 그림 10-11에 보여 주었다. 세번째 단계는 *T*2와 a를 합쳐서 *T*3을 만드는데 무게는 10+8=18이다.

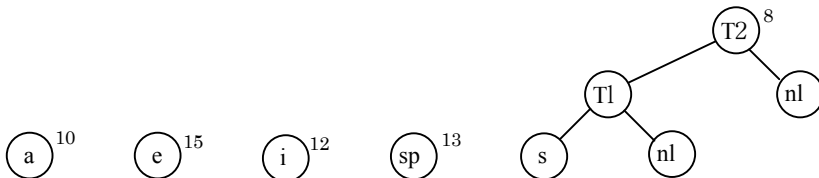



그림 10-11. 두번째 결합을 한후의 하프만알고리듬

그림 10-12는 이 조작의 결과를 보여 주고 있다.

그림 10-13은 이 나무들이 어떻게 뿌리 $T4$를 가진 새 나무에 합쳐 지는가를 보여 준다. 다섯번째 단계에서 뿌리 e와 $T3$을 가진 나무를 결합하는데 이 나무들은 가장 작은 무게를 가지게 된다. 이 단계의 결과를 그림 10-14에 보여 주었다.

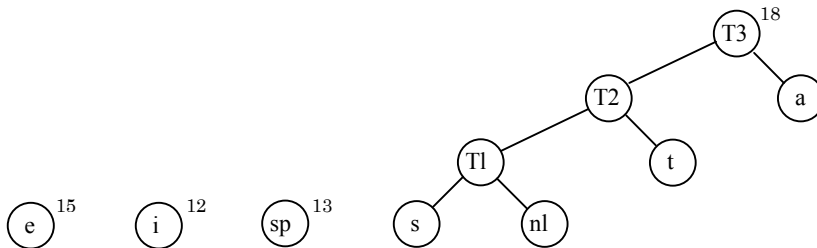



그림 10-12. 세번째 결합을 한후의 하프만알고리듬

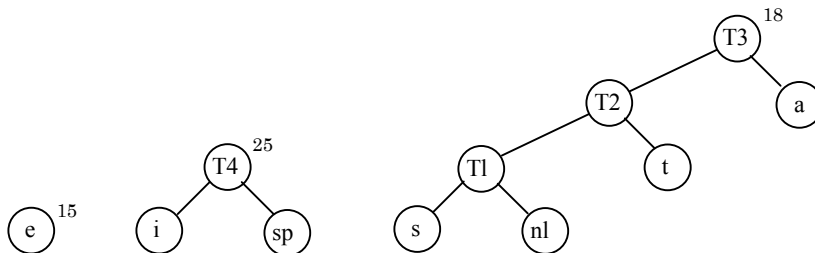



그림 10-13. 네번째 결합을 한후의 하프만알고리듬

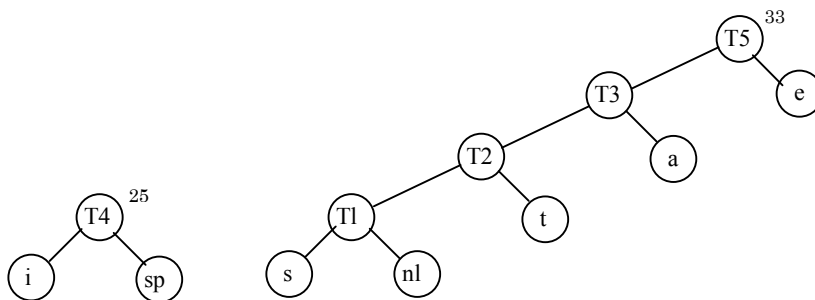



그림 10-14. 다섯번째 결합을 한후의 하프만알고리듬

마지막으로 최적나무를 그림 10-8에 보여 주었는데 이것은 남아 있는 두 나무를 합침으로써 만들어 진다. 그림 10-15는 뿌리가 $T6$인 최적나무를 보여 준다.

하프만알고리듬이 최적부호를 만들어 낸다는것을 증명하는데 련관된 더 구체적인 내용은 련습문제에서 보도록 하자. 여기에서는 먼저 이미 채워 넣지 못한 나무가 어떻게 자라는가를 보았으므로 나무가 충분히 채워 져야 한다는것을 귀유법으로 증명하는것은

어렵지 않다. 다음으로 가장 작은 출현빈도수를 가지는 두개의 기호 α와 β가 가장 깊은 두개의 매듭으로(다른 매듭들이 깊다 해도) 되여야 한다는것을 보자.

이것은 또한 모순에 의한것으로 보기 쉽다. 왜냐하면 α나 β가 가장 깊은 매듭이 아니라고 하면 거기에는 어떤 γ (나무가 찼다는것을 상기하시오.)가 있어야 하기때문이다. 만일 α가 γ 보다 출현수가 더 적으면 나무에서 그것들을 서로 바꾸어 부호길이를 개선할수 있다.

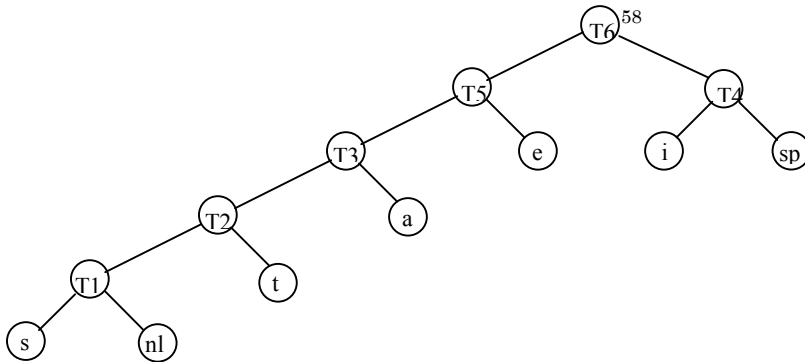



그림 10-15. 마지막결합을 한후의 하프만알고리듬

또한 같은 깊이에 있는 임의의 두 매듭에 있는 문자들은 최적화에 영향을 주지 않으면서 서로 바뀔수 있다는것을 론증할수 있다. 이것은 최적나무가 언제나 가장 적은 출현수를 가진 두개의 기호를 형제처럼 포함하게 된다는것을 보여 준다. 이렇게 하여 첫 단계에서는 오유가 없었다.

증명은 귀납법으로 완성할수 있다. 나무를 합칠 때 새로운 문자모임을 뿌리에 있는 문자로 고찰한다. 이렇게 실례에서는 4번째까지의 합치기를 한후 e와 $T3$과 $T4$로 이루어지는 문자모임을 볼수 있다. 이것이 증명의 가장 기교적인 부분으로 되며 모든 세부적인데 이르기까지 충분히 지적할것을 요구한다.

이것이 **탐욕**알고리듬으로 되는 리유는 매 단계에서 전반적인 상태를 고려함이 없이 합치기를 진행하였기때문이다. 여기에서는 다만 가장 작은 2개의 나무를 선택한다.

우선권대기렬에서 나무를 가지고 있고 무게에 의해 순서화되였다면 실행시간은 $O(C\log C)$이다. 그것은 C개이상의 요소들을 더 가지지 않는 우선권대기렬에 대하여 하나의 buildHeap연산, $2C-2$개의 deleteMin연산 그리고 $C-2$개의 insert연산을 가지기때문이다. 련결목록을 리용하여 **우선권대기렬**을 간단히 실현하면 $O(C^2)$알고리듬을 얻는다. 우선권대기렬을 실현하는데서 알고리듬의 평가는 C가 얼마나 큰가에 관계된다. 전형적인 아스키문자모임인 경우에 원소 C는 두제곱실행시간보다 더 적게 걸린다. 그러한 응용프로그람은 가상적으로 모든 실행시간이 디스크입출력요구시간에 관계되며 이것은 입력파일을

읽기 및 출력파일을 쓰기할 때 압축된 형태로 할것을 요구한다.

여기에서 강조할것이 두가지 있다. 첫째는 부호화정보가 압축파일의 머리부에 있어야 하는데 그렇게 하지 않으면 복호화가 불가능하다는것이다. 이렇게 하는데는 여러가지 방법이 있는데 련습문제 10-4를 고찰해 보자. 작은 파일에 대해서 이 부호표를 전송하는데 걸린 시간은 압축하는데 걸리는 시간에 비하면 무시할만큼 적으며 결과는 압축된 파일이 복호되여야 한다. 물론 이것은 검사해 볼수 있으며 초기의 부호는 완전히 그대로 재현된다. 큰 파일에 대한 부호표의 크기는 그렇게 크지 않다.

앞에서 이야기된 두번째 문제는 두단계로 알고리듬이 처리된다는것이다. 즉 첫번째 단계는 자료의 출현수를 구하는것이며 두번째 단계는 부호화를 진행하는것이다. 이것은 명백히 큰 파일을 처리하는 프로그람에 대하여 매우 효과가 있는 알고리듬이다. 일부 다른 방법들은 참고서들에 서술되여 있다.

3. 근사상자채우기문제

여기에서는 **상자채우기**(*Bin packing*)문제를 푸는 몇가지 알고리듬을 고찰한다. 이 알고리듬들은 속도가 빠르기는 하나 반드시 최적풀이를 만들어 내는것은 아니다. 그러나 얻어 진 풀이가 최적풀이와 근사하다는것을 증명하려고 한다.

크기가 $s_1, s_2, \ldots, s_N$인 N개의 항목이 있다고 하자. 모든 크기는 $0 < s_i \leqq 1$을 만족한다. 문제는 이 항목들을 가장 적은 수의 상자에 채우는것이다. 여기서 매 상자는 단위용량을 가진다고 한다. 실례로 그림 10-16에서는 크기 0.2, 0.5, 0.4, 0.7, 0.1, 0.3, 0.8을 가진 항목목록들을 최적으로 묶는 한가지 방법을 주었다.

상자채우기문제에는 두가지 부류가 있다. 첫 부류는 **직결상자채우기**(*on-line bin packing*)이다. 여기서는 매 항목이 다음번 항목이 처리되기전에 상자에 배치되여야 한다. 두번째 부류는 **비직결상자채우기**(*off-line bin packing*)문제이다. 여기서는 모든 항목의 입력이 끝나기전에는 아무 작업도 할수 없다. 이 두가지 부류의 차이는 제8장 제2절에서 설명하였다.

직결알고리듬

먼저 **직결알고리듬**이 항상 최적풀이를 줄수 있는가, 지어 계산제한이 없는 경우에도 그렇게 할수 있는가 하는것을 고찰하자. 계산제한이 없는 경우라도 직결알고리듬은 다음번 항목이 처리되기전에 하나의 항목을 배치해야 하며 그 배치는 변경시킬수 없다.

직결알고리듬이 항상 최적풀이를 줄수 없다는것을 보여 주기 위해 특별히 풀기 어려운 자료들을 주자. 무게가 $1/2-\varepsilon$인 M개의 작은 항목들에 이어서 무게가 $1/2+\varepsilon$인 M개의 큰 항목들의 입력렬 I_1을 보자($0<\varepsilon<0.01$). 매 상자에 이 항목들을 한개의 작은것

과 한개의 큰것을 함께 놓으면 M개의 상자로 채워진다는것은 명백하다. 이 채우기를 할 수 있는 최적직결알고리듬 A가 있다고 하자. 무게 $1/2-\varepsilon$인 M개의 작은 항목으로만 이루어 진 입력렬 I_2에 대한 알고리듬 A의 동작을 보자. I_2는 $\lceil M/2 \rceil$개의 상자에 채울수 있다. 그런데 A는 매 항목을 개별적인 상자안에 배치한다. 이로부터 I_2에서도 A가 I_1의 첫 절반에서 한것과 같은 결과를 내기때문에 I_1의 첫 절반은 정확히 I_2와 같다. 이것은 A가 I_2에 대한 최적인 수의 2배만한 상자를 리용한다는것을 의미한다.결과는 직결상자채우기 알고리듬에서 최적인 풀이는 없다는것을 증명하고 있다.

우에서의 증명이 보여 주는것은 직결알고리듬은 입력이 언제 끝나는지 전혀 모르며 따라서 알고리듬전반에서 그 매개의 집행은 알고리듬에 의해 매 순간마다 해야 한다는것을 알수 있다. 우의 방법으로 다음과 같은것을 증명할수 있다.

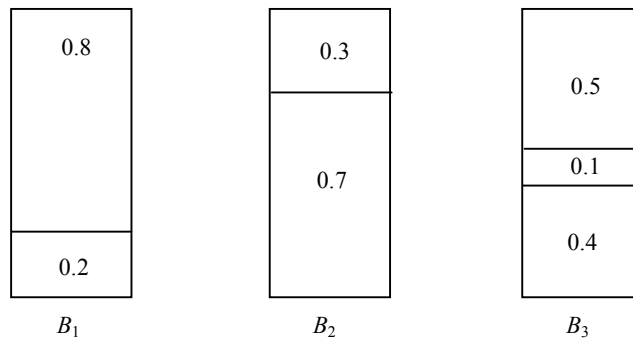



그림 10-16. 0.2, 0.5, 0.4, 0.7, 0.1, 0.3, 0.8항목들에 대한 최적포장

정리 10-1.

임의의 **직결상자채우기**알고리듬에는 적어도 최적상자수를 $\frac{4}{3}$만큼 리용하게 하는 입력렬이 존재한다.

증명:

그렇지 않다고 가정하고 간단화를 위하여 M이 짝수라고 가정하자. 우에서 본 입력렬 I_1에서 동작하는 임의의 직결알고리듬 A를 보기로 하자. 우에서 이 입력렬은 M개의 큰 항목에 뒤이어 M개의 작은 항목으로 이루어 졌음을 다시 상기시킨다. M번째 항목을 처리한후 알고리듬 A의 동작을 보자. A가 이미 b개의 상자를 리용했다고 하자. 알고리듬의 이 시점에서 상자의 최적수는 $M/2$이다. 왜냐하면 매 상자에 두개의 항목을 배치하였기때문이다. 이렇게 $\frac{4}{3}$보다 더 좋은 실행담보를 가정한 조건에서 $2b/M < \frac{4}{3}$라는것을 알수 있다.

모든 항목이 채워진후에 알고리듬 A의 동작을 보자. 모든 상자는 b번째 상자가 정확히 하나의 항목을 포함한후에 창조된다. 왜냐하면 모든 작은 항목들이 첫 b개 상

자에 배치되기때문에 두개의 큰 항목은 하나의 상자에 들어 갈수 없을것이다. 이로부터 첫 b개의 상자가 거의 두개씩 가지고 있고 그리고 남아 있는 상자가 하나씩 가지고 있으므로 채워진 $2M$개 항목이 적어도 $2M$-b개의 상자를 요구할것이다. $2M$개의 항목은 M개의 상자를 리용하여 최적으로 채울수 있으므로 우리의 실행담보는 $(2M-b)/M<\frac{4}{3}$이라는것을 보증하게 된다.

첫번째 착오는 $b/M<\frac{2}{3}$, 두번째 착오는 $b/M>\frac{2}{3}$라는것을 암시하며 이것이 하나의 모순이다. 이렇게 직결알고리듬은 최적상자수를 $\frac{4}{3}$보다 더 적은수로 채울것이라는 담보가 없다.

사용된 상자수가 최적수의 두배보다 많지 않다는것을 담보하는 간단한 3개의 알고리듬이 있다. 또한 좀 더 복잡하지만 더 좋은 담보를 주는 알고리듬도 있다.

다음적합

대체로 가장 간단한 알고리듬은 **다음적합**(*next fit*)방식이다. 어떤 항목을 처리할 때 그것이 제일 마지막항목을 배치하는 그 상자안에 배치가 적합한가를 검사한다. 만일 적합하면 거기에 배치하고 그렇지 않으면 새 상자를 만든다. 이 알고리듬은 실현하기 쉽고 선형적인 시간안에 동작한다.

그림 10-17은 그림 10-16과 같은 항목에 대하여 만들어 지는 상자채우기방법을 보여 준다. 뿐아니라 다음적합방식은 프로그람이 간단하며 해석도 쉽다.

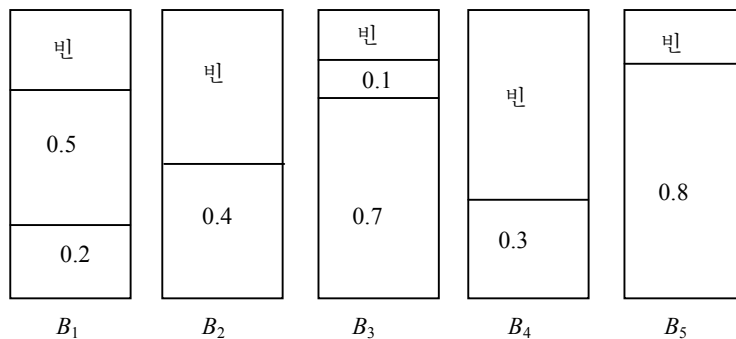



그림 10-17. 항목 0.2, 0.5, 0.4, 0.7, 0.1, 0.3, 0.8에 대한 다음적합방식

정리 10-2.

항목목록 I를 채우는데 요구되는 최적상자수를 M이라고 하자. 다음적합방식은 $2M$개 이상의 상자를 쓰지 않는다. 다음적합방식은 $2M$-2개의 상자를 사용하도록 하는 입력렬이 존재한다.

증명:

임의의 이웃상자 B_j, B_{j+1}이 있다고 하자. B_j와 B_{j+1}안의 모든 항목의 크기의 합은 1보다 커야 한다. 그것은 그 크기의 합이 1보다 작으면 이 모든 항목들이 B_j에 배치되기때문이다. 가령 이 결과를 이웃한 모든 상자쌍에 응용하면 빈 공간의 거의 절반이 절약된다. 이렇게 다음적합은 기껏해서 두배의 최적상자수를 리용한다.

이 한계를 확정하기 위해 i가 홀수이면 N개의 항목이 s_i=0.5만한 크기를 가지고 i가 짝수이면 s_i=2/N만한 크기를 가진다고 가정하자. 그리고 N이 4로 나누어 진다고 하자. 그림 10-18에서 보여 준 최적상자수는 N/4개를 포함하고 매개는 크기가 0.5인 2개의 요소들을 포함한다.

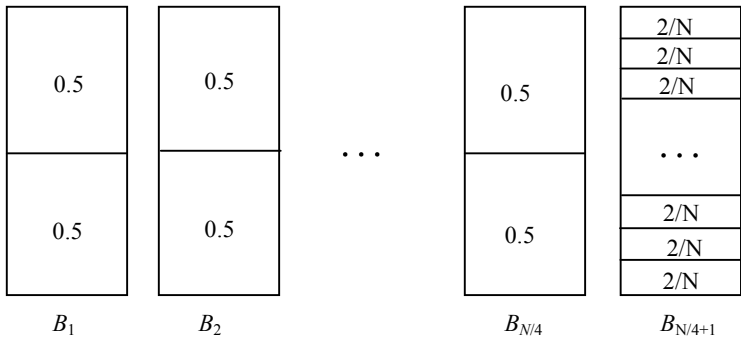



그림 10-18. 항목 0.5, 2/N, 0.5, 2/N, 0.5, 2/N…에 대한 최적채우기

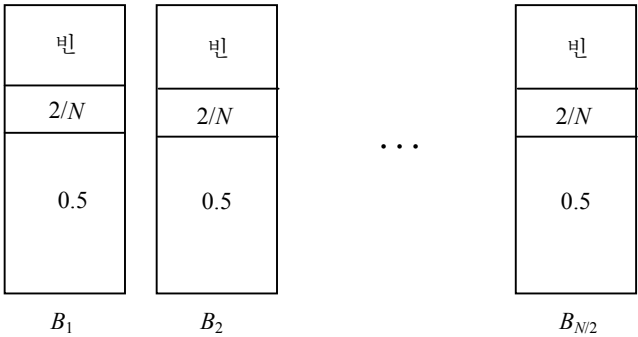



그림 10-19. 항목 0.5, 2/N, 0.5, 2/N, 0.5, 2/N…에 대한 다음적합포장

그리고 하나는 크기가 2/N인 N/2개의 요소들을 포함하며 총 상자수는 (N/4)+1로 된다. 그림 10-19는 다음적합방식이 N/2개의 상자수를 리용한다는것을 보여 준다. 이렇게 다음적합방식은 최적수의 거의 두배만한 상자수를 리용한다.

처음적합

다음적합방식이 담보가 있다 해도 사실 질 좋은 알고리듬이 되지 못하는 리유는 상

자를 만들어야 할 필요가 없을 때에도 새 상자를 만들기때문이다. 실례에서 새로운 상자를 만들기보다는 크기가 0.3인 항목을 B_1 혹은 B_2에 배치하는것이 오히려 더 좋다.

처음적합(*first fit*)방식의 전략은 순서대로 상자를 탐색해 나가며 그것을 배치하는데 충분히 큰 첫 상자안에 새 항목을 배치하는것이다. 이렇게 새 상자는 앞선배치결과가 다른 빈 공간을 남기지 않을 때에만 만들어 진다. 그림 10-20은 표준입력에 대한 채우기결과를 보여 준다.

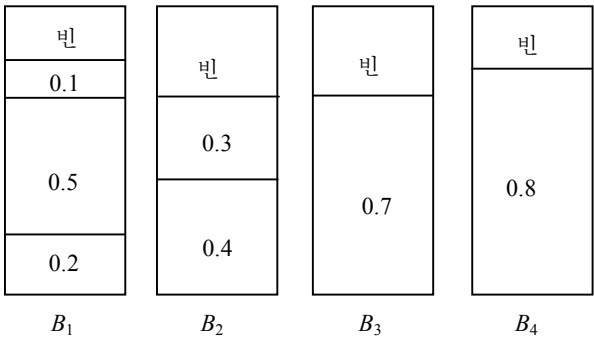



그림 10-20. 항목 0.2,0.5, 0.4, 0.7, 0.1, 0.3, 0.8…에 대한 처음적합

처음적합방식을 실현하는 간단한 방법은 상자목록을 련속적으로 탐색하여 매 항목을 처리하는것이다. 이 처리는 $O(N^2)$만큼한 시간을 요구한다. 이것은 처음적합방식으로 $O(N\log N)$만한 시간동안 동작을 허용하는데 이 실험은 여기서 피하고 련습문제에서 보도록 한다.

중요한것은 임의의 점에서 최대한 하나의 상자가 절반이상 빈다는것을 확신할수 있고 이로부터 만일 두번째 상자가 역시 절반이 비였다면 첫번째 상자는 꽉 찼다는것을 말해 준다. 이렇게 직결적합방식은 최적상자수와 거의 두배만한것으로 상자를 리용한다는 담보를 할수 있다.

한편 다음적합방식의 실행한계를 증명하는데 리용되였던 적합치 못한 경우들은 처음적합방식에 적용하지 않는다. 이로부터 더 좋은 한계가 얻어 질수 있겠는가를 생각할수 있는데 대답은 긍정적이지만 증명은 복잡하다.

정리 10-3.

항목들의 목록 I를 묶는데 요구되는 최적상자수를 M이라고 하자. 처음적합방식에서 쓰려는 상자수는 $\left\lceil \frac{17}{10}M \right\rceil$을 넘지 않는다. 처음적합방식이 상자수 $\frac{17}{10}(M-1)$을 리용하는 입력렬이 반드시 존재한다.

증명:

이 장의 마지막에 있는 참고문헌을 보시오.

처음적합방식이 이전의 정리에서 지적한것만큼 그렇게 충분하지는 못하나 이러한 실례를 그림 10-21에 보여 주었다.

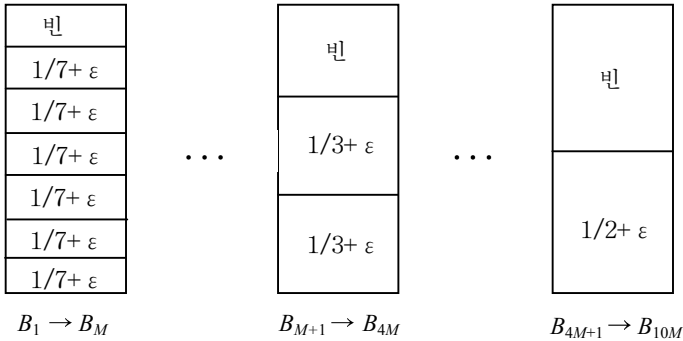



그림 10-21. 처음적합방식인 경우 상자수는 $6M$개 대신 $10M$개 리용

입력은 크기가 $1/7+\varepsilon$인 $6M$개의 항목, 련달아서 $1/3+\varepsilon$인 $6M$개로 이루어 져 있고 뒤이어 또 크기가 $1/2+\varepsilon$인 $6M$개의 항목이 있다. 하나의 간단한 상자채우기에서는 하나의 상자안에 매개 크기가 같은것을 하나씩 넣어 $6M$개가 필요하다. 처음적합에서는 $10M$개가 필요하다.

처음적합이 0과 1사이에 정규분포된 큰 항목들로 배치될 때 경험적인 결과는 처음적합방식에서의 대략적인 최적상자수보다 2% 더 많이 소비된다는것을 보여 준다. 많은 경우에 이것은 충분히 가능한것이다.

최적적합

세번째 직결방법은 **최적적합**(***Best fit***)이다. 찾은 첫번째 자리에 새 항목을 배치할대신 모든 상자중에서 가장 적당한 자리에 항목들을 배치한다. 전형적인 상자묶기를 그림 10-22에 보여 주었다. 크기가 0.3인 항목을 B_2대신 B_3에 채우면 통에 완전히 들어 맞게 된다. 이것은 상자수의 선택방안을 더 좋게 할수 있다는데로부터 실행담보가 개선될것이라는 기대를 가질수 있다.

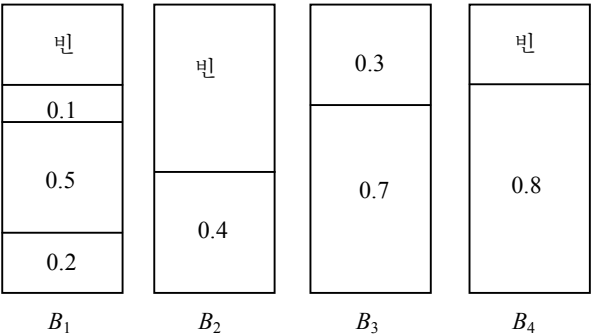



그림 10-22. 항목 0.2, 0.5, 0.4, 0.7, 0.1, 0.3, 0.8에 대한 최적적합방식

최적적합방식은 최적인 경우에 1.7배정도로 되므로 거의가 경계에 도달한다고 볼수 있다. 그밖에도 최적적합방식은 프로그람이 간단하며 특히 알고리듬의 계산에 $O(N\log N)$ 만큼한 계산량밖에 들지 않으며 우연적인 입력렬에 대해서 더 좋게 실행된다.

비직결알고리듬

해답을 얻기전에 전체 항목목록을 볼수 있다면 더 좋게 실행될것이라고 예측할수 있다. 사실 최적으로 채울수 있는 상자채우기를 얻어 냈기때문에 이미 우에서 본 직결인 경우의 알고리듬은 리론적으로 개선된것이다.

모든 직결알고리듬에서 중요문제는 큰 항목들을 채우기가 어려운것인데 특히 입력때에야 결심이 생긴다는것이다. 기본방도는 제일 큰 항목들을 처음에 배치하면서 그 항목들을 정렬하는것이다. 그다음 **내리처음적합**(*first fit decreasing*)과 **내리최적적합**(*best fit decreasing*)이라는 알고리듬들을 리용하는 최적적합방식과 처음적합방식을 각각 적용할수 있다. 그림 10-23은 이 경우에 이것이 최적풀이를 준다는것을 알수 있다(물론 이것은 완전히 옳은것은 아니다.).

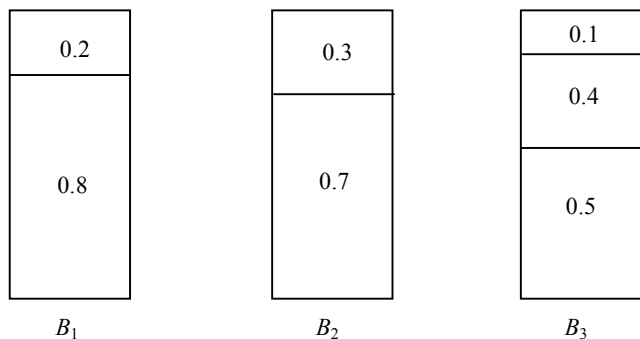



그림 10-23. 항목 0.8, 0.7, 0.5, 0.4, 0.3, 0.2, 0.1에 대한 처음적합방식

이 부분에서는 **내리처음적합방식**(*first fit decreasing*)을 취급한다. 내리처음적합방식에 대한 결과들은 거의 같다. 항목의 크기가 명백하지 않으므로 일부 사람들은 이 **비중가처음적합**알고리듬을 리용하는것을 더 좋아한다. 입력항목의 크기는 일반성을 잃지 않으면서 이미 정렬되였다고 가정할수 있다.

먼저 강조할것은 $6M$개의 상자대신에 $10M$개의 상자를 써서 처음적합을 실현하는 적합치 못한 경우에는 항목들이 정렬되였을 때에도 적용하지 않는다는것이다. 가령 최적적인 채우기가 M개의 상자를 리용한다면 내리처음적합방식알고리듬은 최대한 $(4M+1)/3$개 이상의 상자를 쓰지 않는다.

결과는 두가지 관찰에 관계된다. 첫번째는 $\frac{1}{3}$보다 큰 무게를 가진 항목들모두는 첫 M상자에 놓는다. 이것은 무게가 $\frac{1}{3}$이하인 항목들은 그 나머지(여분)상자들에 놓게 된다

는것을 의미한다. 두번째는 여분상자들에 있는 항목들의 수는 기껏해서 M−1일수 있다. 이 두가지 결과를 묶으면 커서 [(M−1)/3]개의 여분의 상자가 요구될수 있다. 이제 이 두가지 관측결과를 증명한다.

보조정리 10-1.

입력크기가 $s_1, s_2, \ldots, s_N$(작아 지는 순서로 분류된)인 N개의 항목이 있다고 하자. 그리고 최적채우기수는 M개라고 하자. 그러면 내리처음적합방식이 여분의 상자에 배치하는 모든 항목수는 기껏해서 $\frac{1}{3}$만한 크기를 가진다.

증명:

i번째 항목이 상자 +1에 처음으로 배치되였다고 가정하자. $S_i \leqq 1/3$이라는것을 증명해야 한다. 이것을 귀유법으로 증명하려고 한다. $S_i > 1/3$이라고 가정하자.

크기가 정렬된 순서에 따라 배렬되였으므로 $S_{1,}\ S_2 \cdots,\ S_{i-1} > 1/3$이다. 여기로부터 모든 상자들 $B_{1,}\ B_2 \cdots,\ B_M$은 매개가 기껏해서 두개씩의 항목들을 가진다는 결론을 내릴수 있다. i−1번째 항목이 상자에 배치된후 그러나 i번째 항목이 배치되기전의 체계의 상태를 보자. 현재 첫 M개 상자가 다음과 같이 배렬된다는것을 증명하려고 한다($S_i > 1/3$이라는 가정하에). 우선 몇개의 상자들에는 정확히 하나의 항목요소를 가지고 있으며 다음 남아 있는 상자는 두개의 요소들을 가진다.

$1 \leqq x < y \leqq M$이고 B_x는 두개의 항목, B_y는 하나의 항목을 가진 두개 상자 B_x, B_y가 있다고 가정한다. x1과 x2는 B_x에 있는 2개 항목이며 y_1는 B_y에 있는 항목이다. $x_1 \geqq y_1$이므로 x_1는 상자에 더 쉽게 배치된다. 이와 류사한 방법으로 $x_2 \geqq s_i$라고 쓸수 있다. 그러므로 $x_1 + x_2 \geqq y_1 + s_i$이다. 이것은 s_i가 B_y에 배치될수 있다는것을 시사해 준다. 가정에 의하여 이것은 불가능하다. 그래서 만일 $S_i > 1/3$이면 s_i를 처리하려고 할 때 첫 M개 상자에서 첫 j개 상자는 하나의 항목을 가지고 다음 M-j상자는 두개의 항목을 가지도록 배치된다.

보조정리를 증명하는데서 M개의 상자에 모든 항목을 배치할 방도는 없다는것을 보게 될것이다. 이것은 보조정리의 증명을 어렵게 한다.

명백하게 두개의 항목들 $S_{1,}\ S_2 \cdots,\ S_j$은 어떤 알고리듬에 의하여 하나의 상자에 배치될수 없다. 그래도 그것들을 배치하려 한다면 처음적합방식으로는 그렇게 하기가 어려울것이다. 또한 처음적합방식이 첫 j개 상자에 크기가 $S_{j+1}, S_{j+2} \cdots,\ S_j$인 임의의 항목들을 배치하지 못하며 이 방법은 적합치 못하다. 이로부터 임의의 채우기에서 특히 최적채우기에서 이 항목들을 포함하지 않는 j개의 상자가 있어야 한다는것이다. 이러한 결과는 크기 $S_{j+1}\ S_{j+2,}\ \cdots,\ S_{j-1}$의 항목들은 M-j개 상자들의 일부 모임에 포함되여야 하며 또한 앞에서 고

찰한 내용으로부터 그런 항목들의 총수는 2(M-j)[28]이라는것이다.

이것은 만일 S_i>1/3이면 M개 상자중의 어느 하나에도 S_i를 배치할 방도는 없다는것을 강조하면서 증명을 끝낸다. 명백히 이것은 j개 상자들중 어느 하나에도 배치할수 없으며 이로부터 이것이 배치가 가능하면 처음적합은 풀이를 가지는것으로 된다. 이것을 여분의 M-j개 상자들의 하나에 배치하자면 2(M-j)+1개의 항목들을 M-j개의 상자에 분배해야 한다. 그러므로 일부 상자는 세개의 항목을 가지도록 해야 한다. 또 이 매개는 1/3보다 더 큰것을 가지는것으로 되는데 명백히 이것은 모순이다.

이것은 사실 모든 크기를 가진 항목들이 M상자에 배치될수 있다는것을 반박한다. 그러므로 이 기본가정은 맞지 않는다. 이로부터 S_i≦1/3이다.

보조정리 10-2.

여분의 상자에 배치되는 대상의 수는 기껏해서 M-1이다.

증명:

여분의 상자에 적어도 M개의 대상들이 있다고 가정하자. M개의 상자에 대상모두를 포함시킬수 있다는데로부터 $\sum_{i=1}^{N} s_i \le M$ 라는것을 알수 있다. 상자 B_i는 $1 \le j \le M$에 대하여 총무게 W_j로 채워 진다. 첫 M개 여분의 대상들이 $x_1, x_2, \ldots, x_M$의 크기를 가지고 있다고 하자. 그다음 첫 M상자안의 항목에 첫 M개 여분의 항목을 더하여 모든 항목의 부분모임을 이루는데 다음의 식으로 쓸수 있다.

$$\sum_{i=1}^{N} s_i \ge \sum_{j=1}^{M} Wj + \sum_{j=1}^{M} x_i \ge \sum_{j=1}^{M} (Wj + x_i)$$

W_j+x_j>1이기때문에 한편 x_i에 일치되는 항목은 B_j,에 놓이게 될것이다. 그러므로

$$\sum_{i=1}^{N} s_i > \sum_{j=1}^{M} 1 > M$$

이다.

그러나 이것은 N개 항목이 M개 상자에 채울수 있다면 불가능하다. 그러므로 기껏해서 M-1의 여분의 항목이 있을수 있다.

정리 10-4.

항목의 목록 I를 채우는데 요구되는 상자수(최적)를 M개라고 하자. 다음 내리최적적합방식은 (4M+1)3이상의 상자수를 쓰지 않는다.

---

[28] 처음적합방식은 M-j 상자에 이 요소들을 배치하며 매개의 상자에는 그 항목들이 배치된다. 그러므로 2(M-j)항목들이 있다.

증명:

M-1개의 여분의 항목이 있는데 이 항목의 크기는 커서 1/3이다. 이렇게 최대한 $[M$-1]/3]개의 여분의 상자들이 있을수 있다. 내리처음적합방식에 의하여 리용되는 상자의 총수는 대략 [(4M-1)/3]≦(4M-1)/3이다.

이것은 내리처음적합방식과 내리다음적합방식 두가지에 대하여 가장 엄밀한 림계점을 제공하는것으로 된다.

정리 10-5.

항목들의 목록 I를 채우는데 요구되는 상자들의 최적수가 M이라고 하자. 그러면 내리처음적합방식은 $\frac{11}{9}M+4$ 이상의 수를 쓰지 않는다. 여기에는 내리처음적합방식이 $\frac{11}{9}M$ 개의 상자들을 리용하는 입력렬이 반드시 존재한다.

증명:

웃한계는 대단히 복잡한 분석을 요구한다. 아래한계는 크기가 $\frac{1}{2}+\varepsilon$ 인 6M개의 요소들, 련이어 $\frac{1}{4}+2\varepsilon$ 의 크기를 가진 6M개의 요소들, 련이어 크기가 $\frac{1}{4}-2\varepsilon$ 인 12M개의 요소들로 이루어 진 입력렬에 대하여 계산해 낸다. 그림 10-24에서 최적채우기는 9M개의 상자들을 요구하지만 내리처음적합방식으로서는 11M의 상자수들을 요구한다는것을 보여 준다.

실지로 내리처음적합방식은 실행이 어렵지 않다. 만일 크기들이 단위크기간격으로 고르게 선정되였다면 나머지상자의 기대수는 $\Theta(\sqrt{M})$ 이다. 상자채우기는 단순한 **탐욕알고리듬**(greedy)이 좋은 결과를 줄수 있다는것을 깨우쳐 주는 산 실례이다.

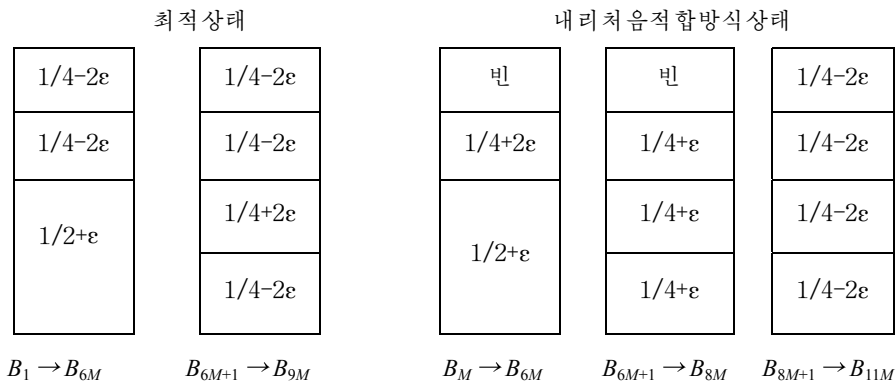



그림 10-24. 내리처음적합방식은 11M개의 상자를 요구하지만 최적방식은 9M개의 상자를 요구한다.

제2절. 분할과 통치

알고리듬을 설계하는데서 리용되는 다른 하나의 일반적수법은 **분할통치(** *Divide and Conquer)*기술이다. 분할통치알고리듬은 두개의 부분으로 구성되여 있다.

분할: 가장 작은 부분의 문제들을 재귀적으로 풀게 된다(물론 기초적인 경우를 제외하고).

통치: 본래의 문제에 대한 풀이는 부분문제들에 대한 풀이로부터 통합하여 구성한다.

전통적으로 이 책은 적어도 두개의 재귀적인 호출 즉 분할, 통치알고리듬이라고 하는 두 방법을 호출하는 루틴들을 써왔다. 이 책의 루틴들은 하나의 루틴을 반복호출하는 것이 아니라 두 루틴을 다 리용한다. 일반적으로 옹근문제를 비본질적인 부분문제들로 분해하는 과정을 고찰한다. 이 책에서 본 일부 재귀알고리듬을 보기로 하자.

이미 여러가지 분할과 통치알고리듬을 취급하였다. 제2장 제4절 3에서 **최대부분순서합문제**(*maximum subsequence sum problem*)에 대한 $O(N\log N)$시간량을 가진 풀이를 보았다. 제4장에서는 선형시간에 나무를 순회하는 방법을 보았다. 제7장에서는 분할과 통치의 가장 고급한 실례로서 병합정렬과 고속정렬방법에 대하여 보았는데 이것들은 각각 최악의 경우와 보통의 경우 $O(N\log N)$의 시간한계를 가진다.

또한 대체로 분할과 통치알고리듬으로 구분할수 없는 재귀알고리듬에 대한 여러가지 실례들도 고찰하였지만 이것은 비교적 간단한 경우이다. 제1장 제3절에서는 수를 출력하는 간단한 프로그람을 보았다. 제2장에서는 유효자리수를 포함하는 수를 취급하는 재귀호출방법에 대하여 보았다. 제4장에서는 2진탐색나무에서 간단한 탐색을 실현하는 루틴들을 실험하였다. 제6장 제6절에서는 제일 왼쪽에 있는 더미들을 병합하는데 리용하는 간단한 실례들을 보았다. 제7장 제7절에서는 보통의 경우에 선형시간을 가지는 선택문제알고리듬을 주었다. 분리모임에 대한 Find연산은 제8장에서 재귀적으로 서술되였다. 제9장에서는 **딕스트라**(*Dijkstra*)알고리듬에서 최단경로를 얻어 내는 루틴들과 그라프에서 깊이우선탐색을 하는데 필요한 루틴들을 주었다. 이 알고리듬들에서는 실제로 하나의 재귀호출만이 실행되였을뿐 분할통치알고리듬은 리용되지 않았다.

제2장 제4절에서 보았지만 피보나치수를 계산하는 실용적이지 못한 재귀루틴도 있었다. 이것은 분할통치알고리듬을 리용할수 있지만 문제들이 실지로 분할하여 쓸수 있도록 부분문제로 나누어 져 있지 않았으므로 효과가 없다.

이 절에서는 분할과 통치전략에 대한 많은 실례들을 고찰하게 된다. 첫 응용대상은 **계산기하학**(*computational geometry*)에 대한 문제이다. 평면에 N개의 점이 주어 지고 $O(N\log N)$시간내에 가장 가까운 점들의 쌍이 주어 졌다고 한다. 련습문제들에서는 분할과

통치합에 의하여 풀수 있는 계산기하학문제들도 주고 있다. 최악의 경우에 $O(N)$시간동안에 선택문제를 풀어 내는 하나의 알고리듬을 보자. $2N$비트의 수들을 곱하는데 걸리는 연산시간은 $O(N^2)$으로 증대되며 두개 행렬의 $N*N$연산시간은 $O(N^3)$으로 증대된다는것을 알수 있다. 그러나 이 알고리듬들이 변환알고리듬들보다 최악의 경우 더 좋은 실행시간을 주지만 방대한 입력자료들에 대해서는 확실한 담보가 없다는것이다.

1. 분할통치알고리듬의 실행시간

가장 효과적인 분할과 통치알고리듬은 문제들을 부분문제들로 가르는데 그 매개 문제들은 기본문제의 부분문제들이며 마지막결과를 합쳐서 계산하는데 리용되게 된다. 실례로 두개의 문제를 합쳐서 조작하는 방법을 보자.매개 문제는 기본문제를 절반으로 나눈것인데 그것들은 $O(N)$시간량만큼 보충적인 계산을 진행한다. **분할과 통치알고리듬의 실행시간**(*running time of Divide and Conquer*)식은 아래와 같다.

$$T(N)=2T(N/2)+O(N)$$

제7장에서 이 식이 $O(N\log N)$시간에 풀이를 계산한다는데 대하여 보았다. 아래의 정리는 대부분의 분할통치알고리듬의 실행시간을 결정하는데 리용한다.

정리 10-6.

식 $T(N)= T(N)=aT(N/b)+\Theta(N^k)$의 풀이는

$$T(N)=\begin{cases} O(N^{\log_b a}), & a>b^k \\ O(N^k \log N), & a=b^k \\ O(N^k), & a<b^k \end{cases}$$

여기서 $a\geqq 1, b>1$이다.

증명:

제7장에서 **병합정렬**(*mergesort*)에 대한 분석을 하였는바 N은 b의 제곱수이고 따라서 N-b^m으로 한다. $N/b=b^{m-1}$이고 $N^k=(b^m)^k=b^{mk}=b^{km}=(b^k)^m$이다.

$T(1)=1$이고 $\Theta(N^k)$에서 상수인자를 무시하자. 그러면 다음의 식이 얻어 진다.

$$T(b^m)=aT(b^{m-1})+(b^k)^{\mathrm{m}}$$

만일 a^m으로 나눈다면 방정식은 다음과 같다.

$$\frac{T(b^m)}{a^m}=\frac{T(b^{m-1})}{a^{m-1}}+\left\{\frac{b^k}{a}\right\}^m \tag{10-3}$$

이 방정식을 다른 m값에 대하여 적용하면

$$\frac{T(b^{m-1})}{a^{m-1}} = \frac{T(b^{m-2})}{a^{m-2}} + \left\{\frac{b^k}{a}\right\}^{m-1} \quad (10\text{-}4)$$

$$\frac{T(b^{m-2})}{a^{m-2}} = \frac{T(b^{m-3})}{a^{m-3}} + \left\{\frac{b^k}{a}\right\}^{m-2} \quad (10\text{-}5)$$

$$\cdots$$

$$\frac{T(b^1)}{a^1} = \frac{T(b^0)}{a^0} + \left\{\frac{b^k}{a}\right\}^1 \quad (10\text{-}6)$$

식 10-3 ~ 10-6은 식들을 합계하는 일반적인 수법을 리용한다. 가상적으로 왼쪽에 있는 모든 항들을 오른쪽으로 옮겨서 고찰할수 있다.

$$\frac{T(b^m)}{a^m} = 1 + \sum_{i=1}^{m} \left\{\frac{b^k}{a}\right\}^i \quad (10\text{-}7)$$

$$= \sum_{i=0}^{m} \left\{\frac{b^k}{a}\right\}^i \quad (10\text{-}8)$$

이로부터

$$T(N) = T(b^m) = a^m \sum_{i=0}^{m} \left\{\frac{b^k}{a}\right\}^i \quad (10\text{-}9)$$

만일 $a>b^k$이면 그 합은 1보다 더 작은 비를 가진 기하학적인 수렬이다. 이로부터 무한수렬의 합은 어떤 상수로 계산되고 유한합은 또한 어떤 상수에 의해 한계 지어지며 이것을 식 10-10에 적용할수 있다.

$$T(N) = O(a^m) = O(a^{\log_b N}) = O(N^{\log_b a}) \quad (10\text{-}10)$$

$a=b^k$이라면 매개의 항은 합해서 1이다. 이로부터 합은 $1+\log_b N$항들을 포함하며 $a=b^k$는 $\log_b a=k$라는것을 말해 준다.

$$T(N) = O(a^m \log_b N) = O(N^{\log_b a} \log_b N) = O(N^k \log_b N) = O(N^k \log N) \quad (10\text{-}11)$$

마지막으로 $a<b^k$이면 기하합렬의 항들은 1보다 더 크며 제1장 제2절 3에서 두번째 식을 적용한다. 그러면 이 정리의 마지막경우를 증명하는 아래의 식을 얻는다.

$$T(N) = a^m \frac{(b^k/a)^{m+1} - 1}{(b^k/a) - 1} = O(a^m (b^k/a)^m) = O((b^k)^m) = O(N^k) \quad (10\text{-}12)$$

실례에서 병합정렬은 $a=b=2$이고 $k=1$이다. 두번째 경우는 $O(N\log N)$의 결과를 준다.

만일 세개의 문제들을 풀 때 그 매개가 처음크기의 절반이고 $O(N)$의 추가적인 처리를 가지는 풀이들의 결합으로 풀이가 얻어 진다면 $a=3, b=2, k=1$이다. 첫번의 경우를 여기에 적용하면 매개 한계값은 $O(N^{\log_2 3}) = O(N^{1.59})$이다. 풀이를 합치는데 $O(N^2)$의 처리를 요구하는 세개의 절반크기로 된 문제들을 푸는 알고리듬은 $O(N^2)$의 실행시간을 가진다. 이로부터 세번째 경우를 적용할수 있다.

두가지 중요한 경우들은 정리 10-6으로 밝혀 지지 않는다. 련습문제와 같은 좀 더 어려운 문제들을 위해서 다음 정리들을 더 고찰해 보자. 정리 10-7은 앞선정리를 리용한다.

정리 10-7.

식 $T(N)=aT(N/b)+\Theta(N^k \log^p N), a \geqq 1, b>1, p \geqq 0$의 풀이는

$$T(N) = \begin{cases} O(N^{\log_b a}), & a > b^k \\ O(N^k \log^{p+1} N), & a = b^k \\ O(N^k \log^p N), & a < b^k \end{cases}$$

이다.

정리 10-8.

$\sum_{i=1}^{k} \alpha_i < 1$이면 식 $T(N) = \sum_{i=1}^{k} t(\alpha_i N) + O(N)$의 풀이는 $T(N)=O(N)$이다.

2. 최단점문제

먼저 문제에 입력할 자료들은 어떤 평면에 있는 점들의 목록 P로 주어 진다. 만일 $P_1=(x_1,y_1)$이고 $P_2=(x_2,y_2)$이면 P_1과 P_2사이의 유클리드거리는 $\sqrt{(x_1-x_2)^2+(y_1-y_2)^2}$이다. 현실에서는 점의 최단쌍을 구하는것이 자주 요구된다. 두 점이 같은 위치값을 가질수 있는데 이 경우에 그 쌍은 거리가 0인 최단이다.

N개의 점이 있다면 $N(N-1)/2$개의 거리쌍이 존재한다. 아주 간단한 프로그람으로 이러한 모든 점들을 검사할수 있지만 알고리듬계산량은 $O(N^2)$으로 증대된다. 이 방법은 효률이 낮은 탐색이므로 더 좋은 방법을 생각해야 한다.

점들이 x자리표에 의하여 정렬된다고 하자. 이것은 최악의 경우에 $O(N\log N)$의 시간제한을 받는다. 전체 알고리듬에 대한 시간한계를 $O(N\log N)$로 보기때문에 이 정렬은 그닥 복잡하지 않다.

그림 10-25는 몇개의 표본점모임 P를 보여 준다. 점들이 x자리표에 의하여 정렬되므로 점모임을 두 부분 P_L과 P_R로 구분하는 가상적인 수직선을 그릴수 있다. 이것은 충분히 가능하다. 그러면 제2장 제4절 3의 최대순서합문제에서 보았던것과 거의 같은 상태에

놓이게 된다. 최단점은 P_L에 있을수도 있고 P_R에 있을수도 있으며 혹은 한점은 P_L에 다른 점은 P_R에 있을수도 있다. 그 거리를 d_L, d_L, d_C라고 하자. 그림 10-26은 이 점모임의 구분과 이 세거리를 보여 준다.

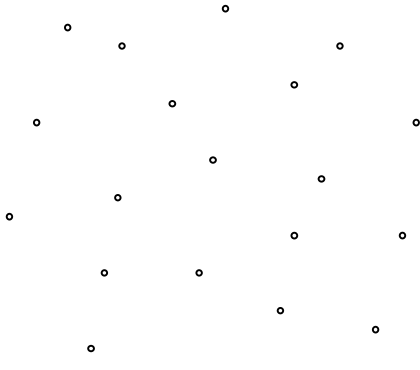



그림 10-25. 몇개의 점모임

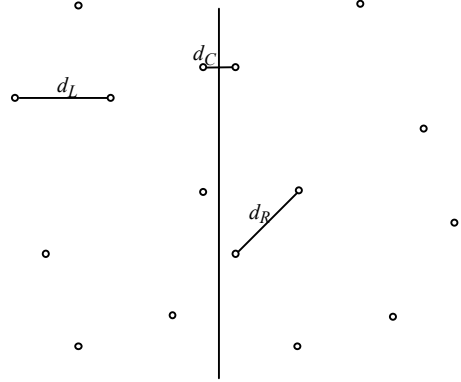



그림 10-26. P점을 P_L과 P_R로 갈라서 최단경로표시

d_L과 d_R는 재귀적으로 구할수 있다. 그다음 문제는 d_C를 구하는것이다. $O(N\log N)$의 풀이를 요구하므로 d_C는 $O(N)$의 추가적인 처리로 계산할수 있어야 한다. 만일 어떤 루틴이 틀의 두개의 절반크기들에 대한 재귀호출과 $O(N)$시간의 추가적인 연산으로 구성되여 있다면 전체 시간은 $O(N\log N)$으로 된다는것을 의미해 보았다.

$\delta=\min(d_L, d_R)$라고 하자. 첫번째로 관측할것은 d_c가 δ로 된다면 다만 d_c만을 계산하는것이다. 만일 d_c가 그 거리라면 d_c를 정의하는 두 점들은 구분선의 δ안에 있어야 한다. 이 구역을 띠라고 하자. 그림 10-27에서 보여 준바와 같이 이 경우에 이 관찰은 고려되여야 할 점의 개수를 제한시킨다(우의 경우 $\delta=d_R$).

d_c를 두가지 계산할수 있다. 균일하게 분포된 큰 점모임에 대하여 띠안에 있는 점의 수는 대단히 작고 사실상 $O\sqrt{N}$ 점들만이 띠안에 평균적으로 있게 된다고 말할수 있다. 이렇게 이 점들에 대하여 계산하면 $O(N)$이라는 수행시간이 걸린다. 프로그람 10-1에서 보여 준 가상부호는 첨수가 0부터 시작된다는 C++의 약속으로부터 이러한 방법이 적용된다.

```
//Points are all in the strip
for( i = 0; i < numPointsInStrip; i++ )
    for( j = i + 1; j < numPointsInStrip; j++ )
        if(dist(pi,pj)< δ )
            δ =dist(pi,pj)
```

프로그람 10-1. Min(δ , d_c)를 계산하는 틀

최악의 경우에 모든 점이 띠안에 있을수 있는데 그러면 이 전략은 항상 선형시간안에 동작할수 없다. 이 알고리듬을 다음과 같이 개선할수 있다. 즉 d_c를 정의하는 두 점들의 y자리표는 최대 δ만큼 차이난다. 만일 그렇지 않으면 $d_c > \delta$이다. 띠의 점들이 그 y자리표에 의하여 정렬된다고 가정하자. 이때 P_i와 P_j의 y자리표의 차가 δ보다 크면 그때 P_{i+1}에로 처리를 넘긴다. 이렇게 갱신한것을 프로그람 10-2에 보여 주었다.

```
// Points are a11 in the strip and sorted by y coordinate
for( i = 0; i < numPointsInStrip; i++ )
    for( j = i + 1; j < numPointsInStrip; j++ )
     if(Pi and Pj 's coordinates differ by more than  δ )
                    break;              // Go to next Pi
                        else   if(dist(pi,pj)< δ )  δ =dist(pi,pj);
```

프로그람 10-2. 재정의된 min(δ, d_c)계산

이 개별적인 검사는 실행시간에 큰 영향을 준다. 그것은 매 P_i에 대하여 P_i와 P_j의 y자리표가 δ보다 큰 차이를 가진다고 확정한 P_i개수는 몇개밖에 되지 않으며 내부의 for순환고리에서 탈퇴하기때문이다. 실례로 그림 10-28에서 보여 준바와 같이 점 P_3에 대하여 두점 P_4와 P_5만이 δ수직거리안의 띠에 놓인다. 최악의 경우에 어떤 점 P_i에 대하여 최대 7개의 P_i점들이 고찰된다. 이것은 이 점들이 띠의 왼쪽 절반의 δ*δ구역 또는 오른쪽 절반의 δ*δ구역에 놓이기때문이다. 다시말하여 δ*δ구역의 모든 점들은 적어도 δ에 의해 분리된다. 최악의 경우에 매 구석에 하나씩 4개 점을 포함하게 된다.

그림 10-27. D_c구역에서 고려되는 모든 점을 포함하는 두 구역띠

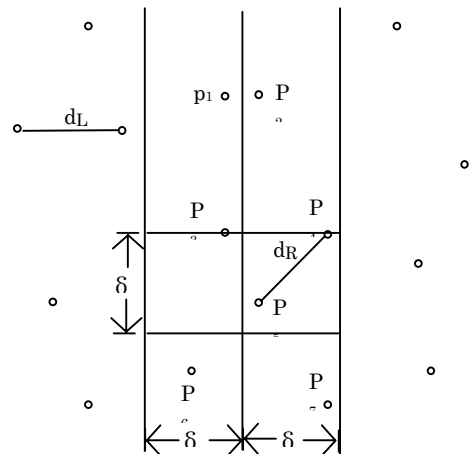



그림 10-28. 두번째 순환에서 고려되는 P_4와 P_5

이 점들중 한점은 P_i인데 최대한 고찰하려는 7개의 점과 떨어 져 있다. 이 최악의 상태를 그림 10-29에 보여 주었다. P_{L2}와 P_{R1}이 같은 자리표를 가지지만 그것들이 서로 다른 점이라는것을 고려하여야 한다.

실제분석에 의하면 λ*2λ인 직4각형구역의 점의 수가 $O(1)$이라는것은 중요하며 명백하다. 매 P_i에 대하여 고찰되는 점이 최대 7개이기때문에 δ보다 작은 d_c를 계산하는 시간은 $O(N)$이다. 그러므로 두개의 결과들 결합시키기 위해 절반크기의 재귀호출에 선형처리를 가하면 $O(N\log N)$의 풀이가 가능하게 된다. 그러나 아직 완전한 $O(N\log N)$풀이를 얻지 못하였다.

문제는 점들의 목록이 y자리표에 의해 구분할수 있다고 가정한것이다. 만일 이 구분을 매 재귀호출마다에서 수행한다면 우리는 $O(N\log N)$의 여분의 작업을 하게 된다. 이렇게 하면 $O(N\log^2 N)$알고리듬이 나온다. 특히 임의의 작업 $O(N^2)$에 비해 볼 때 그리 나쁜것은 아니다. 그러나 $O(N)$에 대하여 매번 재귀호출회수를 감소시키기는 어려우며 $O(N\log N)$알고리듬을 담보하기는 어렵다.

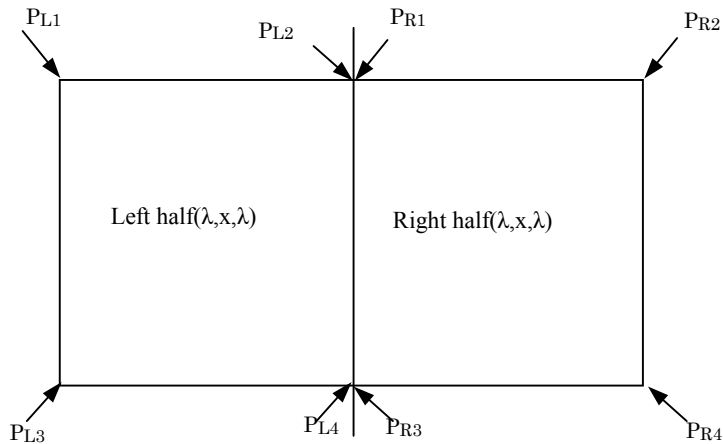



그림 10-29. 직4각형에 배치된 8개 점들, 매개가 2점에 의하여 공유된 두 자리표가 있다.

여기서는 두 목록을 보존해야 한다. 하나는 x자리표에 의해 정렬된 점의 모임이고 다른 하나는 y자리표에 의해 정렬된 점의 목록이다. 이것을 각각 P, Q라고 하자. 이 목록들은 앞의 정렬처리단계에서 얻을수 있으며 따라서 시간한계에 영향을 주지 않는다. P_L과 Q_L은 왼쪽 절반재귀호출로 얻어 진 목록이며 P_R와 Q_R는 오른쪽 절반재귀호출로 얻어 진 목록이라고 하자. 이미 P는 중간에서 쉽게 분리될수 있다는것을 보았다. 분리선이 알려 지기만하면 련속 Q를 통과시켜 Q_L과 Q_R에 요소들을 대처하여 배치할수 있다. Q_L과 Q_R가 자동적으로 y자리표에 의해 정렬된다는것을 쉽게 알수 있다. 재귀호출이 끝나면 Q

목록을 검사하여 띠에 x자리표가 속하지 않는 점들을 제거할수 있다. 그러면 Q는 띠에 속하는 점들만을 포함하게 되며 이 점들은 y자리표에 의해 정렬된다는것이 증명된다.

이 방법을 쓰면 전체 알고리듬은 반드시 $O(N\log N)$으로 된다. 그것은 $O(N)$의 여분의 작업만을 진행하기때문이다.

3. 선택문제

선택문제(*selection problem*)는 N개의 요소의 모임 S에서 k번째 최소요소를 찾는 문제이다. 특별히 흥미 있는것은 중간값을 구하는 특수한 경우이다. 이것은 $k=N/2$일 때 발생한다.

제1장 제6절 7에서 선택문제에 대한 여러가지 풀이를 보았다. 제7장에서의 풀이는 고속정렬의 변종을 리용하여 평균 $O(N)$시간내에 실행된다. 이것은 Hoare의 고속정렬에 대한 초기론문에 서술되여 있다.

이 알고리듬은 보통 평균경우 선형시간에 수행되지만 최악의 경우에는 $O(N^2)$으로 된다. 선택처리는 요소를 정렬함으로써 최악의 경우 $O(N\log N)$시간내에 쉽게 진행되지만 최악의 경우 $O(N)$시간안에 수행되겠는가 되지 않겠는가는 아직 알려 지지 않았다. 제7장 제7절 6에서 본 **고속정렬**(*quicksort*)알고리듬은 실천에서 대단히 효과적이며 리론적으로도 관심이 대단히 큰 알고리듬이다.

기본알고리듬은 간단한 재귀적방법으로 선택하는 전략이라는데 대하여 다시 상기한다. N이 간단히 정렬된 요소들가운데 있는 중단점보다 더 크다고 가정하면 기준값이라고 하는 하나의 요소 v가 선택된다. 나머지원소들은 두개의 모임 S_1과 S_2에 놓여 있다. S_1은 v보다 크기 않은 요소들을 포함하고 S_2는 v보다 작지 않은 요소들을 포함한다. 마지막으로 $k=|S_1|+1$이면 기준값은 k번째로 가장 작은 요소이다. 한편 S에서 k번째로 작은 요소는 S_2에서 $(k-|S_1|-1)$번째로 작은 요소이다. 이 알고리듬과 고속정렬알고리듬의 주요한 차이점은 둘이 아니라 하나의 부분문제만이 존재한다는것이다.

선형알고리듬을 얻기 위해서는 부분문제가 본래문제의 한 부분이며 그러나 단순히 본래문제보다 더 작은 몇개의 요소만이 아니라는것을 알아야 한다. 물론 그렇게 하는데 시간이 걸린다고 해도 그런 원소들을 반드시 찾아 낼수는 있다. 어려운것은 그 기준값을 찾는데 그렇게 많은 시간을 배당할수 없는것이다.

고속정렬에서 기준값를 얻는 좋은 방도는 세개의 원소들을 취하고 그 중간을 선택하는것이라는데 대하여 이미 보았다. 이것은 그러한 기준값이 결과형성에 그런대로 효력을 내기는 하지만 꼭 그렇게 될 담보는 없다. 21개의 요소들을 임의로 선택하고 그것들을 상수시간내에 정렬하여 기준값으로서 11번째로 큰것을 정할수 있으며 이보다 더 좋다고 하는 기준값도 얻을수 있다. 그러나 이 21개 요소들이 21번째로 제일 크다면 기준값은 여전히 제대로 선택되지 못한것으로 된다. 이것을 확장하면 $O(N/\log N)$개의 원소들을 리용하여 $O(N)$의 완전한 시간안에 더미정렬을 진행하며 만족한 점 즉 좋은 기준값을 거의나

확고히 얻을수 있다. 그러나 최악의 경우에 $O(N/\log N)$번째 큰 원소를 선택하며 기준값은 $[N-O(N/\log N)$번째로 큰 요소를 택하게 되므로 이 경우는 적용하지 않는다.

그 기본사상은 역시 리용가치가 있다. 실지로 고속선택이 진행될 때 기대되는 비교수를 개선하기 위하여 그것을 리용할수 있다. 최악의 경우 좋은 시간한계를 구하기 위한 기본사상은 하나이상의 간접준위를 리용하는것이다. 임의의 요소들의 표본에서 중간값을 찾을 대신에 표본에서 중간값을 찾아야 한다.

기본기준값을 선택하는 알고리듬은 다음과 같다.

① N개의 요소들을 여분의 요소(최대로4)들을 무시하면서 5개 요소들의 그룹$\lfloor N/5 \rfloor$으로 갈라 놓는다.

② 매 그룹에서 중간값을 찾는다. 이것은 $N/5$개의 중간값을 가지는 목록 M으로 한다.

③ M의 중간값을 찾는다. 이것을 기준값 v로 되돌린다.

우에서 주어 진 기준값선택규칙을 리용하는 고속선택의 알고리듬을 서술하는데 **5개 요소의 중간중간값**(*median-of-median-of-five*)이라고 표현하자. 5개 부분의 중간전략은 매 재귀보조문제가 최대로 대략 초기의 70%를 차지한다는것을 보기로 하자. 또한 전체 선택알고리듬에 대하여 $O(N)$실행시간을 담보하는데 충분한 속도로 기준값을 계산할수 있다는데 대하여 보자.

N이 5로 완전히 나누어 떨어 지고 그로부터 나머지는 없다고 가정하자. 또한 $N/5$이 홀수이고 따라서 모임 M이 홀수개의 요소를 포함한다고 가정하자. 이것은 이제 보게 되는것처럼 대칭성을 제공한다. 또한 편리상 N이 $10k+5$형태로 되고 모든 요소들은 또한 서로 구별된다고 가정하자. 실제의 알고리듬은 이것이 참이 아닐 때는 조종할수 있게 되여야 한다. 그림 10-30은 $N=45$일 때 기준값이 선택되는 과정을 보여 준다.

그림 10-30에서 v는 알고리듬에 의하여 기준값으로 선택된 요소를 표시한다. v가 9개 요소들의 중심이고 모든 요소들을 분리한다고 가정하였으므로 v보다 큰 4개의 중간값과 v보다 더 작은 4개의 중간값이 존재한다. 그것들을 각각 L과 S라고 표시한다. 큰 중간값(L형태)을 가진 5개 요소를 묶은 그룹을 보자. 그룹의 중간값은 그 그룹의 2개 요소보다는 크고 두개의 요소보다는 작다. H는 큰 요소를 표시한다고 하자. 이것들은 큰 중간값보다는 더 큰 요소들이다.

이와 류사하게 T는 작은 요소들을 표시하는데 이것들은 작은 중간값보다 더 작다. 거기에는 10개의 H형요소들이 있는데 2개는 L형중간값을 가지는 그룹안에 있고 2개 요소들은 v와 같은 그룹안에 있다. 이와 류사하게 10개의 T형요소가 있다. L이나 H형의 요소들은 v보다 크며 S와 T형의 요소들은 v보다 작다는것은 의심할 여지가 없다. 이렇게 문제에는 14개의 큰 요소들과 14개의 적은 요소들이 있다. 따라서 재귀호출은 기껏해서 45-14-1=30개의 요소들에 대하여 진행할수 있다.

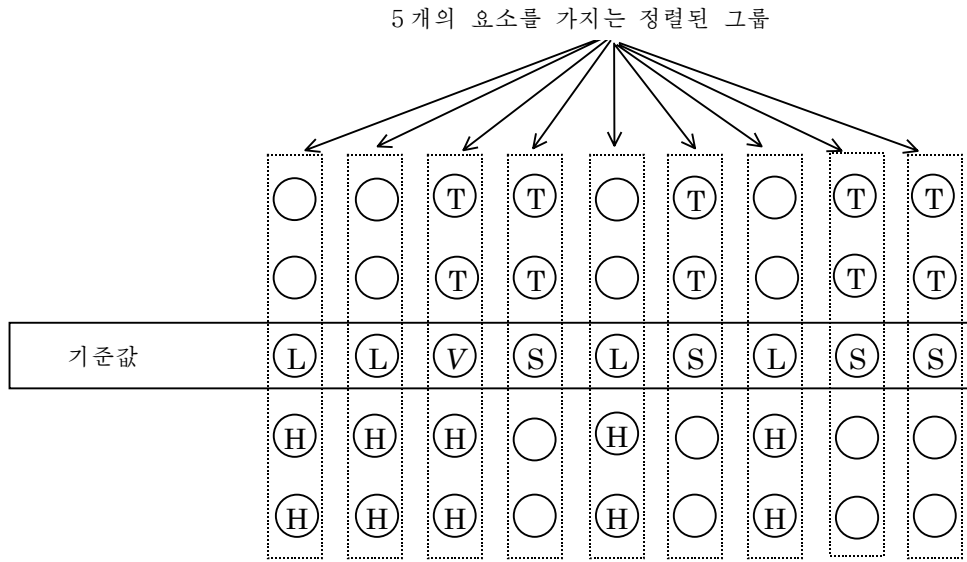



그림 10-30. 기준값선택방법

이러한 분석을 10k+5형태를 가지는 일반적인 N개로 확장하자. 이 경우에 k개의 L형요소들과 k개의 S형의 요소들이 있다. 또한 2k+2개의 H형, 2k+2개의 T형요소가 있다. 이렇게 확고히 v보다 크다고 장담할수 있는 3k+2개의 요소와 확고히 v보다 작다고 장담할수 있는 3k+2개의 요소가 있다. 이런 경우에 재귀호출은 최대 $7k+2<0.7N$개의 요소를 포함할수 있다. N이 10k+5형태가 아니라면 류사한 인수들은 기본결과에 영향을 줌이 없이 처리될수 있다.

기준값요소를 얻는데 걸린 실행시간한계를 계산해 보자. 여기서는 기본적으로 두개의 단계들이 있다. 5개의 요소들의 중간값은 상수시간에 찾을수 있다. 실례로 8번의 비교로서 5개 원소를 정렬해 내는것은 어렵지 않다. 이 공정을 N/5번 진행하여야 하는데 따라서 이 단계는 $O(N)$만한 시간이 걸린다. 다음 N/5개의 요소를 가지는 그룹에서 중간값을 계산하여야 한다. 이것을 수행하기 위한 명백한 방법은 그룹을 정렬하고 그 중간에 있는 요소를 기준값결과로 준다. 그러나 이것은 $O(\lfloor N/5 \rfloor \log \lfloor N/5 \rfloor) = O(N \log N)$ 만한 시간이 걸리므로 이것은 실행하지 않는다. 풀이는 N/5개의 요소에 대하여 선택알고리듬을 재귀적으로 호출하는것이다.

이것으로 기본알고리틈의 작성을 완성한다. 이것을 실제로 실현하자면 자세한 설명이 요구된다. 실례로 처리과정은 정확히 실행되도록 작성되여야 하며 알고리듬이 재귀적호출을 진행할만큼 충분하게 요소그룹을 묶어 주어야 한다. 이를 위한 알고리듬의 실행시간은 대단히 크며 이로부터 알고리듬은 완전히 현실적이지 못하기때문에 더이상 구체적으로 고찰할 필요가 있는 더 상세한 내용을 서술하지 않는다. 리론적인 견지에서 보면 이 알고리듬은 사실 중요한 돌파구라고 말할수 있다. 왜냐하면 다음의 정리가 밝혀 주는

바와 같이 실행시간은 최악의 경우에 선형적이기때문이다.

정리 10-9.
5개 요소의 중간중간값을 리용한 고속정렬알고리듬의 실행시간은 $O(N)$이다.

증명:
이 알고리듬은 $0.7N$과 $0.2N$크기를 가진 두개의 재귀호출들과 선형적인 여분의 처리로 구성된다. 정리 10-8에 의하여 실행시간은 선형적이다.

평균비교수의 감소

분할통치는 또한 선택알고리듬에 요구되는 비교수를 감소시키는데 리용될수 있다. 구체적인 실례를 보자. 1000개의 수의 그룹 S가 있고 100번째로 작은 수를 찾으려고 하는데 그 수를 X라고 가정하자. 100개의 수로 이루어 진 S의 부분모임 S'를 선택하자. X의 값이 S'에서 10번째로 작은 수와 크기상 류사하다고 예측할수 있다. 보다 구체적으로 S'에서 5번째 작은 수는 거의나 X보다 크고 S'에서 15번째로 작은 수는 거의 X보다 더 크다고 가정할수 있다.

보다 일반적으로 S개 요소들의 표본 S'가 N개 요소들로부터 선택된다고 하자. δ를 어떤 처리루틴에서 리용되는 비교의 평균회수를 최소화하기 위해 후에 리용되는 어떤 수라고 하자. 그러면 $v_1=ks/N-\delta$번째와 $v_2=ks/N+\delta$번째로 작은 수는 S'에서 찾을수 있다. S에서 k번째로 작은 수는 거의나 v_1과 v_2사이에 있을것이며 따라서 2δ개의 요소들사이문제에 귀착되게 된다. 낮은 확률로서 k번째로 작은 원소는 이 범위에 있지 않으며 따라서 생각해 보아야 할 문제가 생긴다. 그러나 S와 δ를 잘 선택하면 확률법칙에 의하여 두번째 경우가 전체 작업에 역작용을 하지 않으리라는것을 확증할수 있다. 분석해 보면 $s=N^{2/3}\log^{1/3}N$이고 $\delta=N^{1/3}\log^{2/3}N$일 때 예측되는 비교회수가 $N+k+O(N^{2/3}\log^{1/3}N)$이라는것을 알수 있으며 이것은 더 낮은 순서의 항에 대한 최량예측이다(만일 $k>N/2$이면 $(N-k)$번째로 큰 수를 찾는 문제라고 볼수 있다.).

대부분 분석은 힘들지 않다. 마지막항은 v_1과 v_2를 결정하기 위한 두가지 선택에 드는 계산량을 표시한다. 정렬하는데 걸리는 평균계산시간은 S에서 v_2의 기대값과 N을 더한것과 같다. 즉 이것은 $N+k+O(N\delta/s)$로 된다. 만일 k번째 원소가 S'에 있게 된다면 알고리듬을 끝내는데 걸리는 시간은 S'에서 선택하는데 걸리는 시간 즉 $O(s)$와 같다. 만일 k번째 작은 원소가 S'에 있지 않은 경우 시간은 $O(N)$으로 된다. 그러나 s와 δ가 매우 낮은 확률 $O(1/N)$로 발생한다는 담보하에서 선택한다면 이 확률의 기대시간은 $O(1)$로 되는데 이것은 N이 커지면 0으로 다가 간다는것을 알수 있다. 정확한 계산방법은 련습문제 10-21에 주었다.

이 분석은 중간값을 찾는데 대략적으로 평균 1.5N만한 비교를 하게 된다는것을 보여 준다. 물론 이 알고리듬은 s를 계산하는데 류동소수점수를 리용한다. 이것은 일부 콤퓨터들에서 알고리듬의 실행속도를 떨굴수 있다. 그러나 실험에서 정확히 실행하면 이 알고리듬이 제7장에서 본 고속선택실현에 못지 않다는것을 보여 준다.

4. 산수연산문제들의 리론적개선

여기서는 2개의 N자리옹근수를 곱하는 분할과 통치알고리듬을 서술한다. 이전의 계산모형은 수들이 작기때문에 상수시간에 계산된다고 가정하였다. 큰 수들에 대해서는 이 가정이 그렇게 효력이 없다. 곱하기연산을 하는 동안 수들의 크기견지에서 그 시간을 측정하면 자연적인 곱하기알고리듬은 2차원적인 시간이 걸린다. 분할통치알고리듬으로는 2배의 시간이 걸린다. 또한 두개의 $N*N$행렬곱하기는 기존의 분할통치알고리듬으로 3배이하의 시간이 걸리게 된다.

옹근수곱하기

두개의 N자리 옹근수 X와 Y를 곱한다고 하자. 정확히 X와 Y중 하나가 부수이면 결과도 부수이고 그렇지 않으면 정수이다. 이런 검사를 진행한후에는 $X, Y \geqq 0$라고 가정할수 있다. 손으로 곱하기를 진행할 때 누구나 다 리용하는 알고리듬은 $\Theta(N^2)$연산이 요구된다. 왜냐하면 X의 매 수자들이 Y의 매 수자들에 곱해 지기때문이다.

만일 X=61,438,521이고 Y=94,736,407이면 XY=5,820,464,730,934,047이다. X와 Y를 모든 기호를 포함하면서 최소의 기호들을 가지도록 가르자. 즉 X_L=6,143, X_R=8,521, Y_L=9,473, Y_R=6,407이다. 다음 $X=X_L 10^4+X_R$, $Y=Y_L 10^4+Y_R$라고 하자. 그러면 다음과 같다.

$$XY=X_L Y_L 10^8+(X_L Y_R+X_R Y_L)10^4+X_R Y_R$$

식이 4개의 곱하기 $X_L Y_L$, $X_L Y_R$, $X_R Y_L$, $X_R Y_R$로 구성되고 매 곱하기는 초기문제의 절반크기(N/2수자)로 되여 있다는것을 알수 있다. 10^8과 10^4곱하기는 0을 배치하는것으로 실행한다. 이것과 부분항목들의 더하기는 $O(N)$으로 된다. 이 알고리듬을 리용하여 재귀적으로 4개의 곱하기를 수행한다면 다음의 식을 얻는다.

$$T(N)=4T(N/2)+O(N)$$

정리 10-6으로부터 $T(N)=O(N^2)$임을 알수 있으며 알고리듬을 더는 개선할수 없다. 부분2차원이하의 알고리듬을 얻자면 적어도 4개의 재귀호출을 써야 한다. 기본적으로 고찰할것은

$$X_L Y_R+X_R Y_L=(X_L-X_R)(Y_R-Y_L)+X_L Y_L+X_R Y_R$$

이렇게 10^4의 결수를 구하기 위해 곱하기를 두번이 아니라 한번 진행할수 있다. 표 10-5는 3개의 재귀적인 부분문제를 푸는 방법을 보여 준다.

표 10-5. 분할통치알고리듬의 실제

함수	값	계산량
X_L	6,143	Given
X_R	8,521	Given
X_L	9,473	Given
Y_L	6,407	Given
$D_1= X_L-X_R$	−2,378	$O(N)$
$D_2= Y_R-Y_L$	−3,066	$O(N)$
X_LY_L	58,192,639	$T(N/2)$
X_RX_R	54,594,047	$T(N/2)$
D_1D_2	7,290,948	$T(N/2)$
$D_3=D_1D_2+X_LY_L+X_RY_R$	120,077,634	$O(N)$
X_RX_R	54,594,047	우에서 계산된
$D_3\ 10^4$	1,200,776,340,000	$O(N)$
$X_LY_L10^8$	5,819,263,900,000,000	$O(N)$
$X_LY_L10^8\ + D_3\ 10^4+ X_RY_R$	5,820,464,730,934,047	$O(N)$

재귀방정식은

$$T(N)=3T(N/2)+O(N)$$

을 만족시키며 따라서 $T(N)=O(N^{\log_2 3})=O(N^{159})$을 얻게 된다.

알고리듬을 완성하자면 기본경우를 보아야 하는데 이 경우는 재귀없이 풀수 있다. 두수가 하나의 수자로 이루어 졌다면 곱하기는 표를 보는 방법으로 쉽게 구할수 있다. 만일 한 수자가 0이면 되돌려 주는 값은 0이다. 실제로 이 알고리듬을 리용하려고 한다면 어느것이 콤퓨터에서 더 효과적인가를 보여 주는 기준을 선택하여야 한다.

이 알고리듬은 표준2차원적인 알고리듬에 비하여 더 개선되였다할지라도 드물게 리용된다. N이 작을 때에는 우에서 서술한 알고리듬이 괜찮으며 N이 크면 더 좋은 알고리듬으로 된다. 이 알고리듬에서는 분할과 통치가 광범히 리용된다.

행렬곱하기

기초수값문제는 두 **행렬의 곱하기**(*matrix multiplication*)문제이다. 프로그람 10-3은 $C=AB$를 계산하는 간단한 알고리듬 $O(N^3)$을 주었다. 여기서 A,B,C는 $N\times N$행렬이다. 이 알고리듬은 행렬곱하기의 정의로부터 나온다. $C_{i,j}$를 계산하기 위해서는 A의 i번째 행과 B의 j번째 렬의 곱하기를 진행해야 한다. 보통 첨수는 0부터 시작한다. 식 10-13 은 4개의

수값요소로 다시 묶은 행렬식 AB=C이다.

$$\begin{bmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{bmatrix} = \begin{bmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{bmatrix} \quad (10\text{-}13)$$

```
/* Standard matrix multiplication,
 * Arrays start at 0.
 * Assumes a and b are square. */
matrix<int> operator*( const matrix<int> & a, const matrix<int> & b )
{       int n = a.numrows( );
        matrix<int> c( n, n );
        int i;
        for( i = 0; i < n; i++ )        // Initialization
            for( int j = 0; j < n; j++ )
                c[ i ][ j ] = 0;
                    for( i = 0; i < n; i++ )
            for( int k = 0; j < n; j++ )
                    for( int k = 0; k < n; k++ )
                      c[ i ][ j ] += a[ i ][ k ] - b[ k ][ j ];
                     return c:   }
```

프로그램 10-3. 단순한 $O(N^3)$행렬곱하기

행렬곱하기를 계산하는데 $\Omega(N^3)$계산량이 요구된다는것은 오래동안 가정되여 온 사실이다. 그러나 strassen은 $\Omega(N^3)$의 시간을 타파하는 알고리듬을 내놓았다. ***Strassen*알고리듬**의 기본사상은 매 행렬을 식 10-13에서 보여 준바와 같이 4개의 1/4쪼각으로 나누는것이다. 그러면 쉽게 다음 식이 나온다.

$$C_{1,1}=A_{1,1}B_{1,1}+A_{1,2}B_{2,1}$$
$$C_{1,2}=A_{1,1}B_{1,2}+A_{1,2}B_{2,2}$$
$$C_{2,1}=A_{2,1}B_{1,1}+A_{2,2}B_{2,1}$$
$$C_{2,2}=A_{2,1}B_{1,2}+A_{2,2}B_{2,2}$$

실례로 곱하기 AB를 수행하려면

$$AB = \begin{bmatrix} 3 & 4 & 1 & 6 \\ 1 & 2 & 5 & 7 \\ 5 & 1 & 2 & 9 \\ 4 & 3 & 5 & 6 \end{bmatrix} \begin{bmatrix} 5 & 6 & 9 & 3 \\ 4 & 5 & 3 & 1 \\ 1 & 1 & 8 & 4 \\ 3 & 1 & 4 & 1 \end{bmatrix}$$

8개의 $N/2 \times N/2$ 행렬을 정의한다.

$$A_{1,1}=\begin{bmatrix} 3 & 4 \\ 1 & 2 \end{bmatrix} \quad A_{1,2}=\begin{bmatrix} 1 & 6 \\ 5 & 7 \end{bmatrix} \quad A_{2,1}=\begin{bmatrix} 5 & 1 \\ 4 & 3 \end{bmatrix} \quad A_{2,2}=\begin{bmatrix} 2 & 9 \\ 5 & 6 \end{bmatrix}$$

$$B_{1,1}=\begin{bmatrix}5 & 6\\4 & 5\end{bmatrix}\quad B_{1,2}=\begin{bmatrix}9 & 3\\3 & 1\end{bmatrix}\quad B_{2,1}=\begin{bmatrix}1 & 1\\3 & 1\end{bmatrix}\quad B_{2,2}=\begin{bmatrix}8 & 4\\4 & 1\end{bmatrix}$$

그다음 8개의 $N/2\times N/2$행렬곱하기를 수행하며 $N/2\times N/2$행렬더하기를 수행할수 있다. 행렬더하기는 $O(N^2)$의 시간이 걸린다. 행렬곱하기가 재귀적으로 수행된다면 실행시간은 다음과 같다.

$$T(N)=8T(N/2)+O(N^2)$$

정리 10-6으로부터 $T(N)=O(N^3)$으로 되며 별로 개선을 가져 오지 못하였다는것을 알 수 있다. 옹근수곱하기에서 본바와 같이 부분문제의 수를 8아래로 감소시킬수 있다. strassen은 옹근수곱하기와 류사한 분할통치방식을 리용하였으며 계산식을 효과적으로 배렬하여 7개의 재귀적인 계산식을 리용하였다.

$$\begin{aligned}
M_1&=(A_{1,2}-A_{2,2})(B_{2,1}+B_{2,2})\\
M_2&=(A_{1,1}+A_{2,2})(B_{1,1}+B_{2,2})\\
M_3&=(A_{1,1}-A_{2,1})(B_{1,1}+B_{1,2})\\
M_4&=(A_{1,1}+A_{1,2})B_{2,2}\\
M_5&=A_{1,1}(B_{1,2}-B_{2,2})\\
M_6&=A_{2,2}(B_{2,1}-B_{1,1})\\
M_7&=(A_{2,1}+A_{2,2})B_{1,1}
\end{aligned}$$

곱하기가 수행되면 마지막결과는 다음의 8개의 더하기를 진행하여 얻을수 있다.

$$\begin{aligned}
C_{1,1}&=M_1+M_2-M_4+M_6\\
C_{1,2}&=M_4+M_5\\
C_{2,1}&=M_6+M_7\\
C_{2,2}&=M_2-M_3+M_5-M_7
\end{aligned}$$

이렇게 재치있게 배렬하면 적당한 값들을 얻어 낼수 있음을 증명하는것으로 된다. 그러면 실행시간은 다음의 재귀식을 반복하여 계산함으로써 얻을수 있다.

$$T(N)=TT(N/2)+O(N^2)$$

이 재귀식의 풀이는 $T(N)=O(N^{log_2 7})=O(N^{281})$이다.

보통 고찰되여야 할 구체적인 내용은 N이 2의 제곱수가 아닌 경우를 들수 있다. 그러나 이것은 기본적으로 그렇게 방해되는 내용은 아니다. 스트라센알고리듬은 N이 대단히 크면 간단한 알고리듬들보다는 효력이 떨어 진다는것이다. 그것은 행렬이 성긴행렬인 경우(거의 모두가 0)에 일반화할수 없으며 쉽게 일치되지 않는 류동소수점값을 입력할 때 고전알고리듬에 비해 수값적으로는 덜 안정하기때문이다. 그래서 이런 알고리듬들은

응용이 제한되여 있다. 그러나 이것은 중요한 리론적인 리정표를 주었으며 어떤 문제풀이를 위한 완벽한 알고리듬이 나오기전까지는 비록 정확하지 못하고 어느 정도 복잡하다해도 다른 많은 분야들에서와 마찬가지로 콤퓨터과학에서는 그것을 정확히 증명한다.

제3절. 동적계획법

앞절에서 우리는 수학적으로 재귀적으로 표현되는 문제는 재귀적인 알고리듬에 의하여 풀수 있다는데 대하여 보았다.

일부 재귀적인 수학공식은 직접 재귀적인 알고리듬으로 변환될수 있다. 그러나 많은 경우에 기존사실은 번역프로그람(compiler)이 재귀알고리듬을 정확히 평가할수 없으며 비효률적인 프로그람에 대한 평가도 옳게 할수 없다는것이다. 이런 경우들에 비추어 생각해 보면 재귀적인 알고리듬들은 표에 있는 부분문제에 대한 대답을 체계적으로 기록하는 비재귀알고리듬처럼 재귀알고리듬을 다시 작성해서 번역프로그람에 어느 정도 약간한 도움을 주게 된다. 이 방법을 쓸수 있게 하는 한가지 방법을 **동적계획법**(*dynamic programming*)이라고 한다.

1. 재귀대신 표의 리용

제2장에서 **피보나치수**(*Fibonacci numbers*)를 계산하는 자연적인 재귀프로그람에 대하여 보면서 이 알고리듬이 매우 효률이 적다는데 대하여 알게 되였다. 프로그람 10-5에서 $T(N) \geqq T(N-1)+T(N-2)$를 만족하는 실행시간 $T(N)$을 가진다는것을 상기하자. $T(N)$이 피보나치수와 같은 재귀관계를 만족하며 같은 초기조건들을 가지므로 $T(N)$은 사실 피보나치수와 꼭같은 속도로 늘어 나며 그의 제곱형식으로 된다.

다른 한편 F_N을 계산하기 위하여 필요되는것은 F_{N-1}과 F_{N-2}이므로 바로 그전에 계산된 2개의 피보나치수를 기록하여야 한다. 이에 의한 알고리듬을 프로그람 10-4에 주었다.

```
/**
 * Compute Fibonacci numbers as described in Chapter 1.
 */
int fib( int n )
{
    if( n <= 1 )
        return 1;
    else
        return fib( n - 1 ) + fib( n - 2 );
}
```

프로그람 10-4. 피보나치수를 계산하는데 효률이 없는 알고리듬

재귀알고리듬의 속도가 뜬 리유는 알고리듬이 재귀처리를 반복하기때문이다. F_N을 계산하려면 F_{N-1}과 F_{N-2}를 한번 호출해야 한다. 그러나 F_{N-1}은 재귀적으로 F_{N-2}와 F_{N-3}을 호출하므로 실제로 F_{N-2}를 계산하는데 여러번의 같은 호출이 있게 된다. 전체 알고리듬을 한번 거치면 F_{N-3}도 3번, F_{N-4}는 5번, F_{N-5}는 8번 계산된다는것을 알수 있다. 그림 10-31은 너무 많은 계산이 폭발적으로 증대된다는것을 알수 있다.

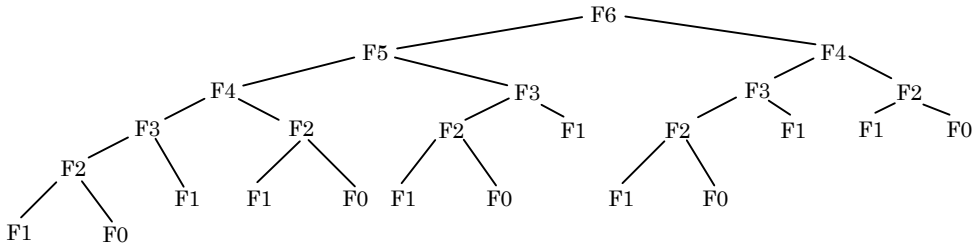



그림 10-31. 피보나치수의 재귀계산나무

만일 콤파일러의 재귀모의알고리듬이 미리 계산된 모든 값들을 보관하고 이미 푼 부분문제에 대해서는 재귀호출을 하지 않게 한다면 이런 지수함수적인 폭발을 피할수 있다. 이로부터 프로그람 10-5에서 보여 준 프로그람이 대단히 효과적이라는것을 말해 준다.

```
/**
 *   Compute Fibonacci numbers as described in Chapter 1.
 */
int fibonacci( int n )
{
                        if( n <= 1 )
                return 1;
        int last = 1;
        int nextToLast = 1;
        int answer = 1;
        for( int i = 2; i <= n; i++ )
        {
            answer = last + nextToLast;
            nextToLast = last;
            last = answer;
        }
      return answer;
}
```

프로그람 10-5. 피보나치수를 계산하는데 선형적인 알고리듬

두번째 실례로서 재귀 $C(N) = (2/N)\sum_{i=0}^{N-1} C(i) + N$, $C(0)=1$ 을 어떻게 푸는가를 제

7장에서 고찰하였다. 구한 수값이 정확한가를 검사하려 한다고 하자. 그러면 프로그람 10-6에 있는 재귀와 동등한 프로그람을 쓸수 있다.

```
double eva1( int n )
{
    if( n == 0 )
       return    1.0;
    Else
    {
       double sum = 0.0;
         for( int i = 0; i < n; i++ )
            sum += eva1( i );
         return 2.0 * sum / n + n;
     }
  }
```

프로그람 10-6. $C(N)=2/N\sum_{i=0}^{N-1}C(i)+N$ 을 평가하는 재귀함수

다시 재귀가 반복되여 호출한다고 하자. 이 경우 실행시간 $T(N)$은 $T(N)=\sum_{i=0}^{N-1}T(i)+N$ 을 만족한다. 왜냐하면 그림 10-32에서 보여 준바와 같이 0부터 N-1까지의 매 수값에 대한 재귀호출이 있고 추가적인 N개의 작업이 있기때문이다(그림 10-32의 나무구조에서 보여 준것과 같다.). $T(N)$에 대한 풀이가 지수함수적으로 증가한다는것을 알수 있다. 표를 리용하여 프로그람 10-7과 같은 프로그람을 얻을수 있다. 이 프로그람은 부차적인 재귀호출을 피하고 $O(N)$시간동안 실행된다. 이것은 완전한 프로그람이 아니다. 그것은 실제로 간단한 변화를 하여 실행시간을 $O(N)$으로 감소시킬수 있기때문이다.

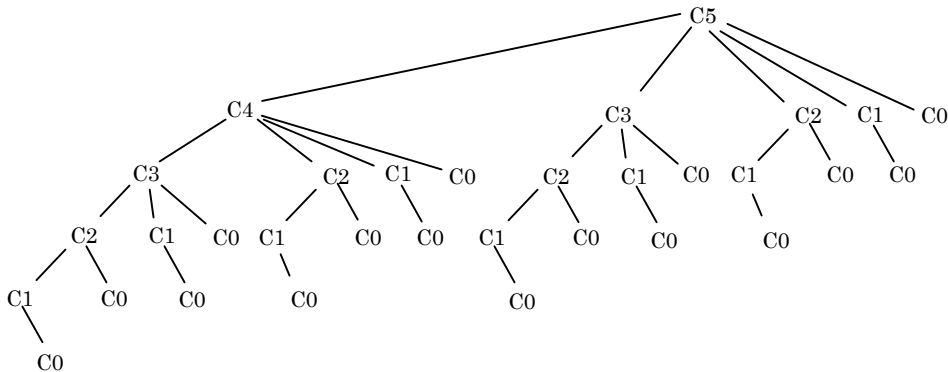



그림 10-32. eval에서 재귀계산나무

```
double eva1( int n )
{
        vector<doub1e> c ( n + 1 );
        c[ 0 ] = 1.0;
        for( int i = 1; i <= n; i++ )
        {
             double sum = 0.0;
           for( int j = 0; j < i; j++ )
                sum += c [ j ];
           c[ i ] = 2.0 * sum / i + i;
        }
        return c[ n ];
 }
```

프로그램 10-7. 표로 $C(N) = 2/N\sum_{i=0}^{N-1}C(i) + N$ 을 평가하기

2. 행렬곱하기의 순서화

차수가 A=50×10, B=10×40, C=40×30, D=30×5인 행렬 A,B,C,D가 주어 졌다고 하자. 행렬곱하기는 바꿈법칙이 성립하지 않지만 련관되여 있다. 즉 행렬곱하기 $ABCD$는 임의의 순서로 괄호로 묶어서 계산하여도 결과는 같다. 크기가 p×q. q×r인 두 행렬을 곱하려면 명백히 pqr번의 스칼라곱하기를 하여야 한다(스트라센알고리듬과 같은 리론적으로 우수한 알고리듬을 리용하여 중요하게는 문제의 변경을 생각할수 없으며 따라서 이 실행한계를 가정하게 된다.). $ABCD$를 계산하기 위해 3번의 행렬곱하기를 수행하는 가장 좋은 방도는 무엇인가?

4개의 행렬인 경우에 철저한 탐색으로 문제를 푸는것은 간단하며 이로부터 다섯가지 곱하기방안을 줄수 있다. 매 경우를 아래에 보여 준다.

- $(A((BC)D))$: BC를 계산하는데는 10×40×30=12,000번의 곱하기가 필요하다. $(BC)D$를 계산하기는 BC를 계산하는데 12,000번의 곱하기가 요구되고 추가적으로 10×30×5=1,500번의 곱하기가 요구된다. 총적으로 13500번이다. $(A((BC)D))$를 계산하는데 $(BC)D$를 위해 13500번, 추가적으로 50×10×5=2,500번의 곱하기 총 16,000번의 곱하기가 요구된다.
- $(A(B(CD)))$:CD계산에 40×30×5=6,000번의 곱하기계산이 요구된다. $B(CD)$를 계산하는데는 CD계산에 6000번과 추가적으로 10×40×5=2,000번 총 8000번 계산한다. $(A(B(CD)))$를 계산하는데는 $B(CD)$계산에 8000번, 추가적으로 50×10×5=2,500번 총 10,500번의 계산이 요구된다.
- $((AB)(CD))$:CD⇒40×30×5=6,000, AB⇒50×10×40=20,000
  $((AB)(CD))$⇒6,000+20,000+50*40*5(=10,000)=36,000
- $(((AB)C)D)$:AB⇒50×10×40=20,000

$(AB)C \Rightarrow 20,000+50 \times 40 \times 30=80,000$

$(((AB)C)D) \Rightarrow 80,000+50 \times 30 \times 50=87,500$

- $((A(BC))D)$:$BC \Rightarrow 10 \times 40 \times 30=12,000$

  $A(BC) \Rightarrow 12,000+50 \times 10 \times 30=27,000$

  $((A(BC))D) \Rightarrow 27,000+50 \times 30 \times 5=34,500$번의 곱하기가 요구된다.

계산결과는 가장 좋은 계산방안이 가장 나쁜 계산방안의 약 1/9만큼 곱하기회수를 줄인다는것을 보여 준다. 이렇게 최적적인 곱하기순서를 결정하는데는 약간한 계산이 필요하다. 그러나 그것을 간단히 하기 위한 명백한 탐욕방식은 없다. $T(N)$을 이 수라고 정의하자. 그러면 $T(1)=T(2)=1$, $T(3)=2$ 그리고 $T(4)=5$이다. 일반적으로는

$$T(N)=\sum_{i=1}^{N-1} T(i)T(N-i)$$

이다.

이를 위해 행렬들이 A_1, A_2,…, A_N이라고 가정하자. 마지막으로 진행된 곱하기는 $(A_1A_2..A_i)(A_{i+1}A_{i+2}…A_N)$이라고 하자. 그러면 가능한 매 i에 대하여 $(A_1A_2..A_i)$를 계산하기 위한 $T(i)$가지 방법과 $(A_{i+1}A_{i+2}…A_N)$을 계산하기 위한 $T(N-i)$가지의 방법이 있다.

이 재귀풀이는 유명한 **까탈로니아**(*Catalan*)수이다. 이 수는 지수함수적으로 증가한다. 이렇게 큰 N에 대하여 가능한 모든 순서짓기를 통하여 지루한 탐색을 할 필요는 없다. 그럼에도 불구하고 이 계산인수들은 지수함수적이라기보다는 본질적으로 더 좋은 풀이를 위한 토대를 제공하고 있다. c_i를 행렬 A에서 $1 \le i \le N$인 렬의 수라고 하자. 다음 A_i는 c_{i-1}의 행수를 가진다. 그것은 곱하기하려는 다음 수는 변화되지 않기때문이다. c_0을 첫 행렬 A_1에서의 행수로 정의하자.

$M_{left,Righ}$는 $A_{left}A_{left+1}\cdots A_{Right-1}A_{Right}$곱하기에 요구되는 곱하기회수라고 가정하자.명백히 $m_{left,left}$=0이다. 마지막곱하기가 $(A_{left}\cdots A_i)(A_{i+1}\cdots A_{Right})$라고 하자. 여기서 $left \leqq i \leqq Right$이라면 리용되는 곱하기수는 $m_{left,1}+m_{i+1,Right}+c_{left-1}c_i\, c_{Right}$이다. 이 세개의 항목들은 $(A_{left}\cdots A_i)$ $(A_{i+1}\cdots A_{Right})$ 그리고 그 적을 계산하는데 요구되는 곱하기의 수를 표시한다.

가령 $M_{left,Right}$를 최적순서짓기에서 요구되는 곱하기의 수가 되도록 정의하고 $Left < Right$ 이면

$$M_{left,Right}=\min_{left \le i < Right}\{M_{left,i}+M_{i+1,Right}+c_{left-1}c_i c_{Right}\}$$

이다. 이 식은 A_{left}, A_{Right}의 최적곱하기배렬을 가진다면 부분문제, $A_{left}\cdots A_i$ 와 $A_{i+1}\cdots A_{Right}$는 부분최적으로 수행될수 없다는것을 암시하고 있다. 이것은 명백하며 다른 한편 부분최적인 계산을 최적계산으로 바꾸어 전체 결과를 개선할수 있다는것을 알수 있다.

공식은 직접 재귀프로그람으로 변환되지만 마지막절에서 본것처럼 그런 프로그람은 효력을 크게 내지 못한다는것이다. 그러나 계산할 필요가 있는 $M_{left,Right}$의 값이 대략 $N^2/2$이기때문에 이런 값들을 보관하는 표를 리용할수 있다는것은 명백하다. 많은 실험들을

통하여 만일 왼-오른=k이면 $M_{left, Right}$을 계산하는데 필요한 $M_{x,y}$값들만이 y-x<k를 만족한다는것을 보여 준다. 이것은 표를 계산하는데 필요한 순서를 알려 준다.

마지막결과 $M_{1,N}$까지의 곱하기의 실제 순서를 추가적으로 알아 내기 위해서는 제9장의 최단경로알고리듬에서의 사상을 리용할수 있다. $M_{left, Right}$을 변화시킬 때마다 영향을 주는 i의 값을 기록하여야 한다. 이 간단한 프로그람을 프로그람 10-8에 준다.

비록 이 장에서 강조되지 않았지만 이것은 많은 프로그람작성자들이 변수이름을 단순한 기호로 짧게 짓도록 한다. C,I,k는 알고리듬작성에 리용한 이름인데 단순기호로 된 변수이다. 그러나 변수이름으로 l을 리용하는것은 될수록 피하여야 한다. 그것은 l은 수자 1과 너무나 비슷하게 보이기때문에 오유가 발생하면 수정하기가 매우 어렵기때문다.

알고리듬의 결과에서 보는바와 같이 프로그람은 3중순환을 하게 되며 $O(N^3)$의 시간동안에 실행된다는것을 쉽게 알수 있다. 참고서에서는 더 빠른 알고리듬을 서술하지만 실제 행렬곱하기를 하는데 걸린 시간이 최적순서화계산시간보다는 여전히 더 크다. 이로부터 이 알고리듬은 여전히 완전히 실천적이다.

```
/**
 * Compute optimal ordering of matrix multiplication.
 * c contains the number of columns for each of the n matrices.
 * c[ 0 ] is the number of rows in matrix 1.
 * The minimum number of multiplications is left in m[ 1 ][ n ].
 * Actual ordering is computed via another procedure using lastChange
 * m and lastChange are indexed starting at 1, instead of 0.
 * Note: Entries below main diagonals of m and lastChange
 * are meaningless and uninitialized.
 */
 void optMatrix( const vector<int> & c,
         matrix<1ong> & m, matrix<int> & lastChange )
 {
     int n = c.size( ) - 1;
     for( int left = 1; left <= n; 1eft++ )
         m[left ][ left ] = 0;
     for( int k = 1; k < n; k++ )     // k is right - left
         for( int left = 1; left <= n - k; 1eft++ )
         {
              // For each position
              int right = left + k;
              m[ left ][ right ] = INFINITY;
              for( int i = left; i < right; i++ )
              {
                  long thisCost = m[ left ][ i ] + m[ i + 1 ][ right ]
                       + c[ left - 1 ] * c[ i ] * c[ right ];
                  if( thisCost < m[ left ][ right ] ) // Update min
                  {
                        m[ left ][ right ] = thisCost;
                        lastChange[ left ][ right ] = i;          }
              }
         }
 }
```

프로그람 10-8. 행렬곱하기의 최적순서를 찾는 프로그람

3. 최적2진탐색나무

두번째 동적계획법의 실례에서는 다음과 같은 입력자료를 고찰한다. 단어들의 목록 $w_1, w_2, \cdots, w_N$ 과 그 단어들의 발생확률 $p_1, p_2, \cdots, p_N$이 주어 졌다. 문제는 기대되는 총 호출시간을 최소로 하는 한가지 방법을 리용하여 이 단어들을 어떤 2진탐색나무에 배렬하는것이다. 2진탐색나무에서 깊이 d에서 어떤 요소를 호출하여 비교하게 되는 비교수는 d+1이고 가령 w_i가 깊이 d_i에 놓여 있다면 $\sum_{i=1}^{N} p_i(1+d_i)$ 를 최소화해야 한다.

실례로서 표 10-6은 몇개의 문장에서 7개의 단어들을 그의 발생확률과 함께 보여 준다. 그림 10-33은 3가지 가능한 2진탐색나무들을 보여 준다. 그것들의 탐색시간들을 표 10-7에 보여 주었다.

표 10-6. 2진탐색나무최적화문제를 위한 자료입력

단어	확률
a	0.22
am	0.18
and	0.20
egg	0.05
if	0.25
the	0.02
two	0.08

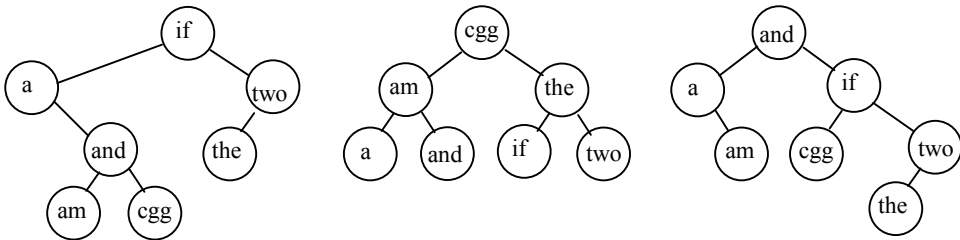



그림 10-33. 표자료에 기초한 3개의 가능한 2진탐색나무

첫번째 나무는 **탐욕법**(*greedy strategy*)을 써서 형성하였다. 호출될 가능성이 제일 큰 단어는 뿌리에 두었다. 왼쪽과 오른쪽 부분나무들은 재귀적으로 형성되였다. 두번째 나무는 정확히 균형이 잡힌 탐색나무이다.

이 문제는 이미 보아 온 **탐욕**(*Greedy*)알고리듬으로 풀수 있는 하프만부호화나무의 구축과 아주 비슷하게 보이기때문에 처음에 놀랄수 있다. 최적2진나무의 구축은 더 힘들다. 왜냐하면 자료가 오직 잎들에만 나타나도록 구축되는것은 아니며 또한 나무는 반드시 2진탐색나무의 속성을 만족해야 하기때문이다.

동적계획법의 해답은 다음의 두가지로 유도할수 있다. 다시한번 2진탐색나무에 정렬된 단어들을 w_{left}, w_{left+1}, $\cdots w_{Right-1}$, w_{Right}순서로 배치한다고 가정하자. 최적2진탐색나무가 $Left \leqq i \leqq Right$인 뿌리 w_i를 가진다고 가정하자. 그러면 왼쪽 부분나무는 반드시 w_{left}, $\cdots$, w_{i-1}을 포함해야 하며(2진탐색나무속성에 의하여) 오른쪽 부분나무는 반드시 w_{I+1}, $\cdots$, $w_{righy}1$을 포함해야 한다.

표 10-7. 세개의 2진탐색나무의 비교

Input		Tree # 1		Tree # 2		Tree 3	
단어	**확률**	**호출량**		**호출량**		**호출량**	
w_i	p_i	Once	Sequence	Once	Sequence	Once	Seq ence
a	0.22	2	0.44	3	0.66	2	0.44
am	0.18	4	0.72	2	0.36	3	0.54
and	0.20	3	0.60	3	0.60	1	0.20
egg	0.05	4	0.20	1	0.05	3	0.15
if	0.25	1	0.25	3	0.75	2	0.50
the	0.02	3	0.06	2	0.04	4	0.08
two	0.08	2	0.16	3	0.24	3	0.24
총합	1.00		2.43		2.70		.15

더구나 이 두 부분나무들은 최적이여야 하며 그렇지 않으면 $w_{left}, \cdots w_{Right}$에 대해서 더 좋은 풀이를 주는 최적부분나무로 교체된다.

이로부터 최적2진탐색나무의 시간 $C_{left,Right}$에 대한 식을 쓸수 있다. 그림 10-34를 참고하시오.

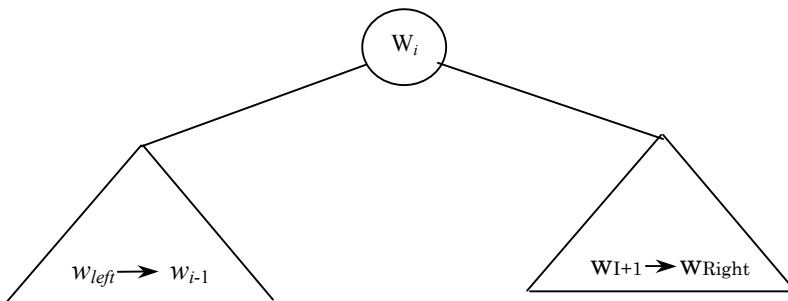



그림 10-34. 최적2진탐색나무의 구조

그림 10-34에서 보여 준바와 같이 매 매듭은 이 부분나무들에서 그것들의 개개의 뿌

리에서 보다 w_i에서 한층 더 깊다. 이로부터 우리는 $\sum_{j=Left}^{i-1} p_j$ 와 $\sum_{j=i+1}^{Right} p_j$ 식을 더 추가해야 한다.

만일 *Left* > *Right*이라면 나무의 탐색시간은 0이다. 즉 이것은 NULL인 경우인데 2진탐색나무에 대해서 항상 이 경우를 가지고 있다. 반대로 뿌리를 탐색하는데 P_i만한 시간이 든다. 왼쪽 부분나무는 그의 뿌리와 련관된 시간 $C_{left,i-1}$을 가지며 오른쪽 부분나무는 $C_{i+1,Right}$를 가진다.

$$C_{Left,Right} = \min_{Left \le i \le Right} \left\{ p_i + C_{Left,i-1} + C_{i+1,Right} + \sum_{j=Left}^{i-1} p_j + \sum_{j=i+1}^{Right} p_j \right\}$$

$$= \min_{Left \le i \le Right} \left\{ C_{Left,i-1} + C_{i+1,Right} + \sum_{j=Left}^{Right} p_j \right\}$$

이 식으로부터 직접 최적2진탐색나무의 시간을 계산하는 프로그람을 작성할수 있다. 일반적으로 실제탐색나무는 $C_{left,Right}$를 최소화하는 *i*값을 보존하는것으로 유지된다. 이 표준재귀루틴은 현재 작용중인 나무를 출력하는데 쓰일수 있다.

표 10-8은 이 알고리듬에 의해 생성되게 되는 표를 보여 준다. 단어들의 매 부분범위에 대한 최적2진탐색나무의 비용과 뿌리가 보존된다. 그림에서 제일 밑에 기입된것은 입력되는 단어들의 전체 모임에 대해서 최적인 2진탐색나무를 계산한 값이다. 최적나무는 그림 10-33에 보여 준 세번째 나무이다.

표 10-8. 간단히 입력된 자료에 대한 최적2진탐색나무계산

Iteration=1	a..a		am..am		And..and		egg..egg		If..if		The..the		two..two	
	.22	a	.18	am	.20	and	.05	egg	.25	If	.02	the	.08	two
Iteration=2	a..am		am..and		And..egg		egg..if		if..the		The..two			
	.58	a	.56	and	.30	and	.35	if	.29	If	.12	two		
Iteration=3	a..and		am..egg		And..if		egg..the		if..two					
	1.02	am	.66	and	.80	if	.39	if	.47	If				
Iteration=4	a..egg		am..if		And..the		egg..two							
	1.17	am	1.21	and	.84	if	.57	if						
Iteration=5	a..if		am..the		And..two									
	1.83	and	1.27	and	1.02	if								
Iteration=6	a..the		am..two											
	1.89	and	1.53	and										
Iteration=7	a..two													
	2.15	and												

개개의 부분범위(례를 들어 *am..if*)를 위한 최적2진탐색나무의 정확한 계산량은 그림 10-35에 보여 준다. 이것은 *am, and, egg*와 *if*를 뿌리에 배치하여 얻은 최소비용나무를 계산하여 얻어 진것이다. 실례로 *and*가 뿌리에 배치되였을 때 왼쪽 부분나무는 *am..am*을 포함하며(이전의 계산에 비하면 시간이 0.18), 오른쪽 부분나무는 *egg..if* (시간이 0.35)을 포함하며 $p_{am}+p_{and}+p_{egg}+p_{if}=0.68$로서 총합은 1.21이다.

이 알고리듬의 실행시간은 $O(N^3)$이고 이것이 실행될 때 3중순환이 얻어 진다. 그 문제를 위한 $O(N^2)$알고리듬은 련습문제에서 주게 된다.

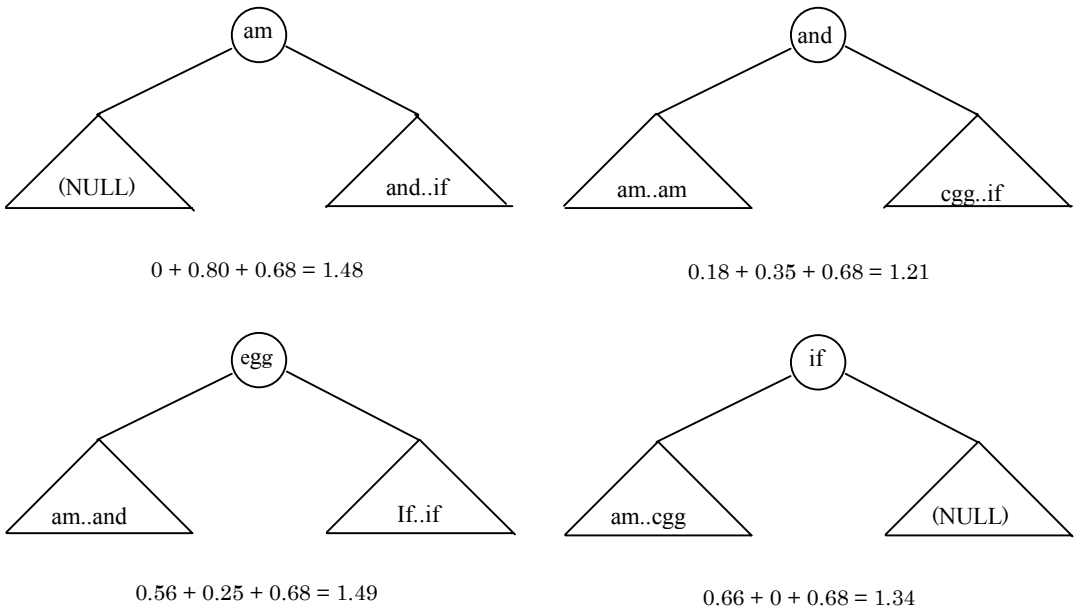



그림 10-35. am부터 if까지의 표전체의 계산

4. 모든 쌍들의 최단경로

세번째이면서 마지막동적계획법응용으로서 방향그라프 $G=(V,E)$에서 **모든 쌍의 최단경로**(*All-pairs shortest path*)를 계산하는 알고리듬을 본다. 제9장에서 어떤 정점에서 다른 모든 점들까지의 최단경로를 찾는 단일원천최단경로문제를 위한 알고리듬을 고찰하였다. 이 알고리듬은 조밀한 그라프우에서 $O(|V|^2)$시간에 실행되나 부분적으로는 성긴 그라프에서 보다 더 빠르다. 조밀한 그라프에서 모든 쌍문제를 푸는 알고리듬을 간단히 보려고 한다. 이 알고리듬의 실행시간은 $O(|V|^3)$이며 딕스트라알고리듬의 $|V|$번의 반복에 접근하는 개선된 방법은 아니지만 대단히 조밀한 그라프에서도 더 빨리 수행된다는것이다. 이 알고리듬은 또한 부의 변들을 가질 때에만 정확히 집행되지만 부의 순환경로는 없다. 이

경우에 딕스트라알고리듬은 실패하게 된다.

딕스트라알고리듬의 중요한 내용에 대하여 다시 상기해 보자(제9장 제3절). 딕스트라알고리듬은 정점 s에서 시작하여 단계별로 실행된다. 그라프의 매 정점은 본질적으로 중간점처럼 선택된다. 만일 현재 선택된 정점이 v라면 매 $w \in V$에 대하여 $d_w=min(d_w, d_v+c_{v,w})$로 설정한다. 이 식은 s로부터 w까지의 최대거리는 이미 알려 진 s로부터 w까지의 거리 즉 s에서 v까지 최적으로 간 결과와 다음 직접 v에서 w까지의 거리라는것을 보여 준다.

딕스트라알고리듬은 동적계획법알고리듬을 위한 사상을 제공해 준다. 우리는정점들을 순차적으로 선택한다. 그리고 v_i에서 중간점으로 v_1, v_2,…, v_k를 리용하는 v_j까지의 최단경로의 무게로서 $D_{k,i,j}$를 정의한다. 이 정의에 의하여 $d_{0,i,j}= c_{i,j}$ 이고 여기서 $c_{i,j}$ 는 (v_I, v_j)가 그라프에서 경계면이 아닐 때 무한대이다. 역시 정의에 의하면 $D_{|k|,j,j}$는 그라프에서 v_i로부터 v_i까지의 경로이다.

프로그람 10-9에 보여 준것처럼 k>0 일 때 $D_{k,I,j}$ 식을 간단히 쓸수 있다. 중간점 v_1, v_2,…, v_k로만 사용되는 v_i부터 v_j 까지의 최단경로는 v_k를 중간점으로 전혀 리용하지 않거나 혹은 두 경로 $v_i \rightarrow v_k \rightarrow$와 $v_k \rightarrow v_i$의 두 경로를 결합한것을 포함하는 최단경로이다. 여기서 두 경로는 매개가 첫 k-1개의 정점들만 중간점으로 리용한다. 이것을 다음 식으로 쓴다. $d_{k,I,j}= min\{ d_{k-1,i,I,} d_{k-1,I,k}+ d_{k-1,k,j} \}$. 이 시간은 역시 $O(|V^3|)$만큼 요구된다. 이전에 본 2개의 동적계획법실례와는 달리 이 시간한계는 다른 때보다 줄어 들지 않고 있다.

다만 k번째 단계는 k-1번째 단계에만 관계되기때문에 두개의 $|V| \times |V|$ 행렬들이 보존될 때만이 그것이 출현한다. 그러나 k에서 시작하거나 끝나는 경로에서 k를 중간점으로 사용하는 경우 거기에 부정순환경로가 있기전에는 결과를 개선할수 없다.

그래서 그것은 $D_{k-1,i,k}=D_{k,i,k}$이고 $D_{k-1,k,j}=D_{k,k,j}$이기때문에 하나의 행렬만이 필요된다. 이것은 오른쪽 항목들의 값을 변화시키지 말고 보존해야 한다는것을 말해 준다. 그러므로 프로그람 10-9의 간단한 프로그람에서처럼 령으로부터 정점들의 수는 C++의 변환으로 확정한다.

```
/**
 * Compute all-shortest paths.
 * a contains the adjacency matrix with
 * a[ i ][ i ] presumed to be zero.
 * d contains the values of the shortest path.
 * Vertices are numbered starting at 0; all arrays
 * have equal dimension. A negative cycle exists if
 * d[ i ][ i ] is set to a negative value.
 * Actual path can be computed using path.
 * NOT_A_VERTEX is -1
 */
```

```
                void allPairs( const matrix<int> & a,
                 matrix<int> & d, matnx<int> & path )
                 {
                   int n = a.numrows( );

                     // Initialize d and path
/*1*/            for( int i = 0; i < n; i++ )
/*2*/                for( int j = 0; j < n; j++ )
                     {
/*3*/                       d[ i ][ j ] = a[ i ][ j ];
/*4*/                       path[ i ][ j ] = NOT_A_VERTEX;
                      }
/*5*/            for( int k = 0; k < n; k++ )

               // Consider each vertex as an intermediate
/*6*/                 for( int i = 0; i < n; i++ )
/*7*/                      for( int j = 0; j < n; j++ )
/*8*/                           if( d[ i ][ k ] + d[ k ][ j ] < d[ i ][ j ] )
                                {
                                     // Update shortest path
/*9*/                                d[ i ][ j ] = d[ i ][ k ] + d[ k ][ j ];
/*10*/                              path[ i ][ j ] = k;
                                 }
                 }
```

프로그램 10-9. 모든 쌍의 최단경로

어떤 완전한 그라프우에서 모든 정점들의 쌍은 련결(쌍방향으로)되며 이 알고리듬을 딕스트라알고리듬의 $|V|$번의 반복보다는 더 빠르다는것이 명백한바 그것은 순환이 빈틈없이 진행되는데서 볼수 있다. 1행부터 4행까지에서는 6행과 10행에서 한것처럼 병렬로 처리될수 있다. 그래서 이 알고리듬은 병렬계산에 아주 적합할것이라고 본다.

동적계획법은 풀이의 시작점을 제공해 주는 아주 강력한 알고리듬설계기술이다. 이것은 본질적으로 더 간단한 문제를 먼저 푸는 분할통치법의 형태변화로서 차이점은 더 간단한 문제에 대해서는 더이상 부분문제로 분할하지 않는것이다. 이로부터 부분문제들은 반복적으로 풀게 되기때문에 그것을 다시 계산하는것보다는 표에 그것들의 풀이를 기록하는것이 더 중요하다. 일부 경우에 풀이는 개선될수 있으며(비록 명백치 못하고 아주 어렵다 할지라도) 일부 다른 경우에 동적계획법기술은 아주 적용하기 쉬운것으로 알려지고 있다.

그런 감각으로 하나의 동적계획법문제를 정통하면 그것들 모두를 정통할수 있다. 동적계획법문제의 보다 구체적인 실례는 련습문제와 참고문헌에서 본다.

제4절 . 란수화알고리듬

매 주 학생들에게 프로그람작성문제를 주는 교원을 생각하자. 교원은 학생들이 자체로 프로그람을 작성하고 있는지 혹은 그들이 제출하는 부호를 리해하고 있는지를 확인해 보고 싶을것이다. 한가지 방도는 프로그람을 제출하는 날에 간단히 질문하는것이고 다른 하나는 그 시험을 방과후시간에 프로그람이 제대로 실행되는지 대충 실행해 본다. 교원인 경우 문제는 질문을 언제 주는가하는것이다.

물론 시험문제를 제출하여 공개하면 시험에 제출되지 않은 문제에 대해서도 50%정도는 대체로 어떤 문제겠는가 하는 판단을 할수 있다. 하나는 교체하게 될 프로그람에 대한 제출방식을 잘 세우는것인데 한편 학생들은 시간이 더 가기전에 이 전략을 궁리해 볼것이다. 다른 하나는 중요하다고 보는 프로그람들에 대한 시험을 조직하는데 시험문제가 학기와 학기사이를 거쳐 류사하게 제출되기 쉽다는것이다. 학생들이 얻은 이 정보 즉 시험전략은 한 학기후에 가치를 잃을것이다.

이런 문제가 없도록 하는 한가지 방법은 동전(coin)을 리용하는것이다. 시험은 매 프로그람에 대하여 치르게 되는데 시험을 시작할 때 선생은 주려고 하는 시험문제가 어떤것인가를 결심하는 동전을 세기 시작할것이다. 이것은 시험을 치기전에 무엇이 나오는지 아는것은 불가능하거나 출현할 문제를 전혀 모르도록 하자는것이다. 그리고 이 시험문제는 학기학기사이에 반복되지 않는다. 그래서 학생들은 미리 시험답안에 대하여 관심함이 없이 시험이 50%확률로 출현할것이라는 기대를 가질것이다.

이 실례는 **란수화알고리듬**(*randomized algorithms*)을 호출하는것으로서 실현한다. 적어서 한번 알고리듬을 실행하는 과정에 하나의 란수가 만들어 져 리용된다. 알고리듬의 실행시간은 개별적인 입력에만 관계될뿐아니라 발생하는 란수에 관계된다.

란수화알고리듬의 최악의 경우의 실행시간은 흔히 비란수발생알고리듬의 최악의 경우의 실행시간과 같다. 중요한것은 좋은 란수발생알고리듬은 입력오유를 내지 않지만 그러나 틀린 란수를 발생(특이한 입력자료에 관계)할수 있다는것이다. 이것은 다만 론리적인 차이일수 있으나 실제로는 아래의 실례에서 보여 주는바와 같이 아주 중요한것이다.

두개의 고속정렬알고리듬들을 생각하자. 알고리듬 A는 기준값으로서 첫번째 요소를 리용하고 알고리듬 B는 기준값으로서 우연적으로 선택된 요소를 리용한다. 두 최악의 경우 실행시간은 $\Theta(N^2)$인데 그것은 매 단계에서의 제일 큰 요소가 기준값으로 선택될수 있기때문이다. 알고리듬 A는 이미전에 정렬된 목록을 주는 연산의 매 회수를 $\Theta(N^2)$시간동안 수행할것이다. 가령 알고리듬 B가 2배의 입력량을 가진다면 란수발생에 관계되는 두 알고리듬의 실행시간의 차이는 역시 2배로 될것이다.

실행시간을 계산해 보면 매개 입력자료들은 아마 같을것이라고 생각하게 된다. 그러나 입력자료가 거의나 정렬되였기때문에 그렇지 않다. 이를테면 특히 고속정렬과 2진탐

색나무처리에서는 통계적으로 기대되는 수자보다는 실행시간이 더 걸린다. 란수화알고리듬을 써서 개개의 자료입력을 진행하는것은 그렇게 소홀히 할 문제가 아니다. 란수들은 중요한것이며 어떤 자료입력을 진행할대신 가능한 란수들로 평균기대시간을 얻을수 있다. 우연적인 기준값들을 써서 고속정렬를 진행하면 기대시간이 $O(N\log N)$인 알고리듬을 얻게 된다. 이것은 언제나 정렬된 자료를 포함하는 임의의 입력에서 실행시간이 $O(N\log N)$으로 기대된다는것을 의미한다. 이 수자는 란수의 통계학에 기초한것이다. 기대되는 실행시간 한계는 대체로 평균한계값보다는 더 크지만 물론 일치하는 최악의 경우의 한계값보다는 더 작다. 다른 한편 우의 선택문제에서 본것처럼 최악의 경우의 한계를 가지는 풀이는 흔히 그것들의 평균경계풀이보다는 실용적이지 못하다. 란수발생알고리듬은 언제나 존재한다.

이 절에서는 란수를 리용하는 두가지 방법을 고찰하게 된다. 첫번째는 탐색기대시간이 $O(\log N)$이며 2진탐색나무연산들을 지원해 주는 새로운 체계들을 보게 된다. 다시말하여 이것은 정확히 나쁜 란수들을 가지는 나쁜 입력이 없다는것을 의미한다. 리론적인 견지에서 보면 이것은 평형이 갖추어 진 탐색나무가 최악의 경우에 이 한계에 도달하기때문에 그리 괜찮은 방법이라고 볼수 없다. 그럼에도 불구하고 란수를 리용하는것은 자료를 탐색하고 삽입하며 특히는 제거하는데서 비교적 간단한 알고리듬을 작성할수 있기때문이다.

두번째 응용은 큰 수들의 원본을 검사하는 란수화알고리듬이다. 이 문제를 푸는데서 다차원시간을 가지는 비란수화알고리듬이 효력이 없다는것은 이미 알려 졌다. 여기에서 보려는 이 알고리듬은 실행이 빠르지만 때때로 오유가 나온다. 오유가 발생할 확률은 대수롭지 않을 정도로 매우 작다.

1. 란수발생기

알고리듬들은 란수를 요구하므로 그것을 만드는 방법을 제기해야 한다. 사실 란수는 콤퓨터에 의하여 가상적으로 만들어 낼수 없고 이 수들은 오직 알고리듬에 의거하기때문에 자유로 생성할수 없다. 일반적으로 이것은 가상란수를 만드는데 충분하며 이 수는 무질서하게 출현하는 수들이다. 란수들은 정적특성으로 알려 진 성질들을 가지고 있는바 가상란수들은 이 성질의 거의 대부분을 만족시킨다.

가령 동전을 세여 나간다고 생각해 보자. 그러면 0(처음에) 혹은 1(마지막에)이 무질서하게 출현하게 된다. 이것을 실현하는 한가지 방도는 체계시간을 리용하는것이다. 시간은 어떤 시점에서부터 초의 수를 세는 옹근수로 기록할수 있다. 다음에 가장 낮은 단위를 리용한다. 문제는 가령 란수렬이 필요될 때에 이것이 잘 만들어 지지 않는다는것이다. 1s는 긴 시간으로 볼수 있으며 시계는 프로그람이 실행되는 동안에 전혀 변화되지 않을수도 있다. 가령 시간이 μ초단위로 기록되였고 프로그람이 그자체에 의하여 실행된다면 발생되는 수렬은 자유도가 낮게 만들어 질수 있다. 이로부터 란수발생기를 호출하는 시

간간격은 매개 프로그람에 대해서 본질상 같아 질수 있다. 다음 실제로 필요되는 란수렬을 얻게 된다.[29] 이 수들은 독자적으로 자유로이 출현한다. 우리가 동전을 뿌릴 때 처음에 앞면이 출현한다면 다음 돈의 세기는 역시 앞면 혹은 뒤면이 자유로 번갈아 출현될 확률이 거의나 같을것이다.

란수를 발생하는 가장 단순한 방법은 선형합성발생기인데 이것은 1951년에 처음으로 레흐머에 의하여 개발되였다.

수 x_1, x_2, . . 은 식

$$X_{i+1}=A\,x_i \bmod M$$

을 만족시키도록 만들어 낸다.

그 란수렬을 시작할 때 어떤 값 x_0이 주어 져야 한다. 이 값을 초기값이라고 한다. $x_0=0$이면 란수렬은 우연성이 떨어 지나 A와 M이 정확히 선정되였다면 임의의 다른 $1<x_0<M$은 균등하게 변한다. 가령 M이 씨수이면 x_i는 결코 0이 아니다. 실례로 $M=11$, $A=7$, 그리고 $x_0=1$이면 란수는 다음과 같이 발생된다.

7, 5, 2, 3, 10, 4, 6, 9, 8, 1, 7, 5, 2,…

$M-1=10$개의 수들 다음에는 란수렬이 다시 반복된다. 그래서 이 렬은 $M-1$의 주기를 가지며 이것은 가능한껏 (비둘기구멍원리에 의하여) 길다. 가령 M이 씨수이면 A는 언제나 $M-1$의 주기를 채우도록 선택된다. 그러나 일부 A의 선택은 그렇게 집행되지 않는다. 가령 $A=5$이고 $x_0=1$이면 란수렬은 5라는 짧은 주기를 가진다.

5, 3, 4, 9, 1, 5, 3, 4,…

M이 길게 즉 31bit길이로 선택되면 주기는 모든 응용에서 굉장히 크게 될것이다. 레흐머는 31bit길이의 $M=2^{31}-1=2,147,483,647$의 리용을 제기하였다. 이것을 위해 $A=48,271$은 옹근주기발생기를 주는 많은 값중의 하나이다.이 값은 이미 전문가들의 고심어린 탐구에 의하여 선택되고 선정되여 온것이다.

이것이 란수를 발생하는 간단한 루틴이다. 일반적으로 클라스변수는 x값렬에서 현재값을 취하는데 리용된다. 란수를 리용하는 프로그람을 수정할 때 아마 리상적인것은 언제나 발생되는 란수렬이 같도록 하기 위하여 $x_0=1$로 선정하는것이다. 프로그람이 동작할 때 체계시계가 리용되든가 아니면 사용자가 초기값을 입력하도록 요구할수도 있다. 이것은 또한 열린간격(0,1)(0과 1은 가능한 값들이 아니다.)사이의 란수실수로 되돌려 주는것이 일반적이다. 즉 이것은 M으로의 나누기를 진행하여 실행될수 있다. 이로부터 임의의 닫긴 간격 $[\alpha,\beta]$사이에서 란수는 표준적으로 계산될수 있다. 이것은 프로그람 10-10에서 《명백한》클라스를 주기는 하지만 유감스럽게도 오유가 있다.

[29] 이제부터 나오는 란수는 가상란수를 의미한다.

```
static const int A = 48271;
static const int M = 2147483647;
class Random
{
  public:
    explicit Random( int initia1Va1ue = 1 );
    int randomIntC,);
    double random0_l( );
    int randomlnt( int low, int high );
  private:
      int state;
  };
/**
  * Construct with initia1Va1ue for the state.
  */
Random::Random( int initialValue )

   {
       if(    initialValue < 0 )
       initia1Va1ue += M;
     state = initia1Va1ue;
     if( state == 0 )
           State = 1;
 }
 /**
 *    Return a pseudorandom int, and change the
 *    internal state. DOES NOT WORK CORRECTLY.
   *    Correct implementation is in Program 10-11.
   */
   int Random::randomint {
      return state = ( A * state ) % M;
   }
   /**
   * Return a pseudorandom double in the open range 0..1
   * and change the internal state.
   */
   double Random::random0_l( )

   {
       return (double) randomlnt( ) / M;
   }
```

프로그램 10-10. 동작하지 않는 란수발생기

이 클라스에서 문제는 곱하기가 자리넘침을 일으키는데 이것이 오유가 없다면 결과와 그에 따르는 가상란수발생에 효과적이다. 스흐라취는 모든 계산은 자리넘침이 없는

32bit처리기로 수행해야 된다고 제기한다. M/A의 상과 나머지를 계산하고 그것들을 Q와 R로 정의한다. 우의 경우 Q=44,488, R=3,399 와 $R<Q$이다. 그러면 다음 식이 성립한다.

$$x_{i+1}=Ax_i \bmod M = Ax_i - M\left|\frac{Ax_i}{M}\right|$$

$$= Ax_i - M\left|\frac{x_i}{Q}\right| + M\left|\frac{x_i}{Q}\right| - M\left|\frac{Ax_i}{M}\right|$$

$$= Ax_i - M\left|\frac{x_i}{Q}\right| + M\left(\left|\frac{x_i}{Q}\right| - \left|\frac{Ax_i}{M}\right|\right)$$

이로부터 $x_i = Q\left\lfloor\frac{x_i}{Q}\right\rfloor + x_i \bmod Q$, 이것을 Ax_i에 도입하면

$$x_{i+1} = A(Q\left|\frac{x_i}{Q}\right| + x_i \bmod Q) - M\left|\frac{x_i}{Q}\right| + M(\left|\frac{x_i}{Q}\right| - \left|\frac{Ax_i}{M}\right|)$$

$$= A(Q) - M)\left|\frac{x_i}{Q}\right| + A(x_i \bmod Q) + M(\left|\frac{x_i}{Q}\right| - \left|\frac{Ax_i}{M}\right|)$$

여기서 $M = AQ + R$ 이며 $AQ - M = R$ 라는것이다. 그러므로 우리는 다음과 같이 쓸 수 있다.

$$x_{i+1}=A(x_i \bmod Q) - R\left|\frac{x_i}{Q}\right| + M(\left|\frac{x_i}{Q}\right| - \left|\frac{Ax_i}{M}\right|)$$

항 $\delta(x_i) = \left\lfloor\frac{x_i}{Q}\right\rfloor - \left\lfloor\frac{Ax_i}{M}\right\rfloor$는 0 혹은 1이며 이로부터 2개의 항들은 옹근수이며 그것들의 차는 0과 1사이에 놓인다. 따라서

$$x_{i+1}=A(x_i \bmod Q) - R\left|\frac{x_i}{Q}\right| + M\delta(x_i)$$

고속검사결과 $R<Q$라는것을 보여 주며 나머지항들은 자리넘침이 일어 나지 않게 계산(이것은 A=48,271로 선택되였기때문에)될수 있다. 더우기 나머지항들이 0보다 작게 평가되면 $\delta(x_i)$=1이다. 이로부터 $\delta(x_i)$는 명백히 계산된것을 필요로 하지는 않지만 간단한 검사로 결정될수 있다. 이것을 프로그람 10-11에서 보게 된다.

```
static const int      A = 48271;
static const int      M = 2147483647;
static const int      Q = M / A;
static const 1nt      R = M % A;
```

```
/**..
 * Return a pseudorandom int, and change the internal state,
 */
int Random::randomlnt( )
{
        int    tmpState = A * ( state % Q ) - R * ( state / Q );
  if ( tmpState >= 0 )
                state = tmpState;
else
                state = tmpState + M;
return state;
}
```

프로그람 10-11. 32bit콤퓨터에서 자리넘침이 일어 나지 않는 란수변경프로그람

이 프로그람은 INT_MAX≧2^{31}-1만큼 길게 란수를 발생한다. 여기서 모든 처리기는 적어도 프로그람 10-11의 표준서고에서 제공된것보다는 더 좋은 란수를 얻어 내려고 시도한다. 그러나 이것은 그렇게 쉽게 해결되지는 않는다. 많은 서고들은 함수에 기초한 란수를 발생하는 발생기를 가지고 있다.

$$X_{i+1}=(Ax_i+C)\bmod 2^B$$

여기서 B는 콤퓨터의 옹근수형에 대응되는 옹근수로 선택되며 C는 홀수이다. 이 발생기들은 언제나 x_i의 값을 만들어 내는데 이 값은 짝수와 홀수를 서로 교번하며 이것은 얻으려는 특성을 만족시키지 못한다. 실지로 2^k주기를 가진 Kbit 주기로 란수들이 발생된다. 다른 란수발생기들은 프로그람 10-11에서 제공된것보다 더 작은 주기로 란수들을 발생한다. 이것들은 긴 란수렬이 요구되는 경우에는 적합치 않다. UNIX drand48함수는 이 형태의 란수발생기로 리용한다. 어쨌든 이것은 48bit로 일치되는 발생기를 리용하며 높은 32bit로 결과를 내보내는것으로서 상수는 A=24,214,903,917, B=48, C=13이다. 마지막으로 식에 이러한 상수들을 대입해 넣음으로써 더 괜찮은 란수발생기를 얻을수 있다고 본다. 실례로 아래의 식은 좋은 란수를 만들어 낸다.

$$X_{i+1}=(48,271x_i+1)\bmod (2^{31}-1)$$

식은 발생기가 무질서하게 만들어 내는 란수식으로서 여기서 초기값이 179,424,105이면

$$[48,271(179,424,105)+1]\bmod(2^{31}-1)=179,424,105$$

이며 란수발생기는 1주기로 순환하게 된다.

2. 건너뛰기목록

먼저 란수화방법을 써서 $O(\log N)$기대시간에 탐색과 삽입 두가지를 진행하게 하는 자료구조를 고찰하자. 이 절의 안내에서 언급한것처럼 이것은 임의의 입력대기렬의 매 연산당 실행시간이 기대값$O(\log N)$을 가진다는것을 의미하는데 이 기대값은 란수발생기에 따른다. 순서짜기조작과 2진탐색나무의 평균탐색시간한계와 일치하는 기대시간한계를 가지는 모든 연산들과 지우기연산을 추가하는것은 가능하다.

탐색하는데 가장 간단하고 적합한 자료구조는 **련결목록**(*linked list*)이다. 그림 10-36은 간단한 련결목록을 보여 준다. 탐색하는데 걸린 시간은 고찰할수 있는 매듭수에 비례하는데 이 수는 기껏해서 N이다.

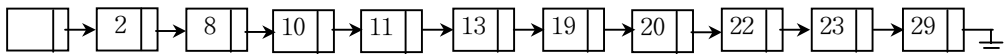



그림 10-36. 간단히 련결한 목록

그림 10-37은 목록에서 매개의 매듭이 두개의 매듭만큼 떨어 진 매듭과 련결된 련결목록을 보여 준다. 이로부터 최대한 $\lfloor N/2 \rfloor$+1개의 매듭들은 최악의 경우로 시험해 볼수 있다.

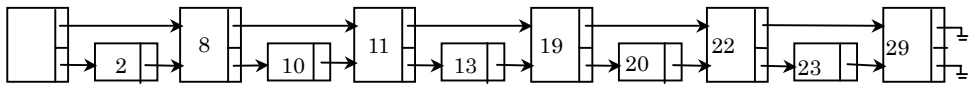
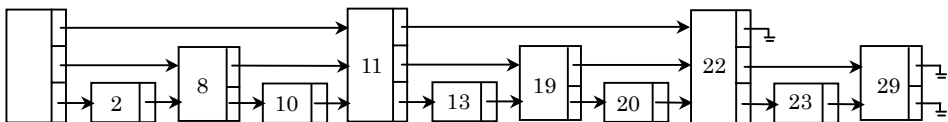



그림 10-37. 2개매듭앞과 련결한 련결목록

이 사상을 확대시켜서 그림 10-38을 얻을수 있다. 여기서 매개 네번째 매듭은 네개의 바로 앞매듭과 련결된다. 여기서 $\lfloor N/4 \rfloor$+2개의 매듭들만 이런 련결을 가지게 된다.

그림 10-38. 4개 매듭앞과 련결한 련결목록

이 알고리듬의 림계점을 그림 10-39에 보여 준다. 매 2^i번째 매듭은 그의 바로 앞 2^i 매듭과 련결된다. 련결되는 총 매듭수는 2의 배수로 된다. 그러나 기껏해서 $\lfloor \log N \rfloor$개 매듭들을 탐색해 나가면서 련결된다.

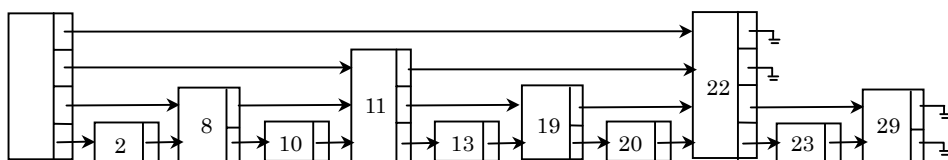



그림 10-39. 2^i매듭앞과 련결한 련결목록

이것은 탐색에 총적으로 $O(\log N)$만큼한 시간이 걸린다는것을 힘들지 않게 알수 있다. 이로부터 탐색은 새 매듭에로 가기 혹은 같은 매듭에서 더 낮은 련결에로 떨구기 등을 포함한다. 이 매개 단계에서 총 탐색시간은 기껏해서 $O(\log N)$을 넘지 않는다. 언급할것은 이 자료구조에서 탐색은 본질적으로 2진탐색이라는것이다. 이 자료구조를 가진 문제는 효과적인 삽입을 하려는 경우에 유연하기 못하다.

이 자료구조를 리용하게 하는 열쇠는 구조조건을 약간 늦추어 주는것이다. 준위 k의 매듭을 k련결을 가지는 매듭이 되도록 정의한다. 그림 10-39에서 보는것처럼 임의의 준위 k의 매듭($k \geq i$)에서 i번째 련결은 적어도 i준위를 가진 다음 매듭에 련결한다. 이것은 련결을 보존하는데 아주 적중한 속성인데 그림 10-39는 이보다 더 제한된 성질을 보여 주고 있다. 따라서 우리는 i번째 련결은 2^i만큼 떨어 진 매듭앞에 련결된다는 제한을 버리고 우에서의 더 적은 제한조건으로 바꾼다.

새 요소를 삽입할 때에는 그에 대한 새로운 매듭을 할당한다. 이 점에서 매듭이 있게 될 준위를 결정해야 한다. 그림 10-39에서 보는바와 같이 대체로 절반정도의 매듭들은 준위가 1인 매듭들이고 1/4정도의 매듭들은 2준위의 매듭들이며 일반적으로 $1/2^i$정도의 매듭들은 i준위의 매듭들이다. 확률분배와 일치하도록 매듭의 준위를 우연적으로 선택한다. 이를 위한 더 쉬운 방도는 첫 머리가 출현할 때까지 동전을 뒤집고 매듭준위와 같이 뒤집은 전체적인 총수를 리용하는것이다. 그림 10-40에 전형적인 건너뛰기목록을 보여 주었다.

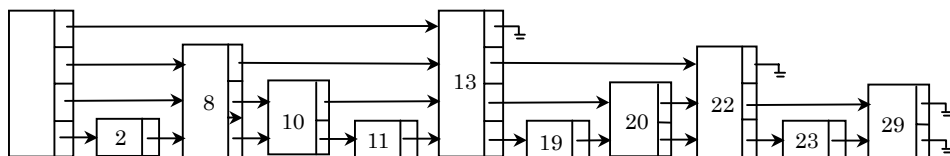



그림 10-40. 건너뛰기목록

이것을 보면 건너뛰기목록알고리듬을 서술하기 쉽다. Find연산을 실현하기 위해 머리부에서부터 제일 높은 련결매듭에서 탐색을 시작한다. 탐색은 이 준위를 따라서 기대

하는것(혹은 *NULL*)보다는 더 긴 다음 매듭을 찾을 때까지 진행한다. 이것을 찾은 다음 더 낮은 준위에 가서 이 과정을 계속한다. 이 조작이 준위 1에서 중지될 때 기대하는 매듭의 앞에 있든지 아니면 그것이 목록에 없을수도 있다. Find에서와 같이 처리하는 Insert를 실현하기 위해서 보다 낮은 준위에로 전환하는 매 점에 대한 자리길을 보존한다. 준위가 우연적으로 설정된 새 매듭은 다음 목록에 잇는다. 이 연산을 그림 10-41에 보여준다.

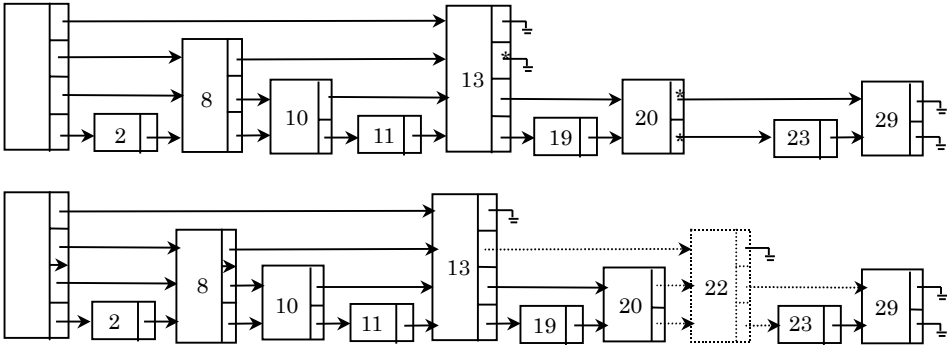



그림 10-41. 삽입하기전과 후의 련결목록

대체로 분석하면 매개 준위에서 기대되는 매듭의 수는 기본알고리듬(비란수화)으로부터 변화되지 않으며 같은 준위에서 매듭에로의 옮김을 수행하는데 기대되는 옮김총수도 변화되지 않는다는것을 보여 준다. 이것은 그런 조작들은 $O(\log N)$기대량을 가진다는 것을 말해 준다. 물론 더 구체적인 검토가 요구되겠지만 이것은 기대량에서 큰 차이가 없다.

건너뛰기목록들은 하쉬표들과 류사하며 여기서 이것들은 목록에 있게 될 요소들의 수를 평가하여야 한다(준위수를 결정할수 있도록). 만일 평가가 변하지 않았다면 더 큰 수를 가정할수 있거나 재해쉬표만들기와 류사한 수법을 쓸수 있다. 실험을 통하여 건너뛰기목록은 균형이 잡힌 많은 탐색나무의 실행만큼 효과가 크며 확실히 많은 언어들의 집행에 가장 쉽게 쓰일수 있는 구조라는것을 알수 있다.

3. 씨수성검사

이 절에서 하나의 큰 수가 씨수인지 아닌지를 결정하는 문제를 시험해 보게 된다. 제2장의 마지막에서 언급된것처럼 일부 **암호학**(*cryptography*)들은 매우 다루기 어려운 큰 수자(200자리수를 두개의 100자리씨수로 나누는)에 관계된다는것이다. 이 체계를 실현하기 위해서는 이 두 씨수를 만들어 내는 방법이 있어야 한다. 이 문제는 리론적으로 d자리수 N이 d차의 다차원적인 사건에 씨수인지 아닌지를 어떻게 검사하는지 누구도 모르

기때문에 아주 흥미를 끄는 문제이다. 가령 3부터 $\sqrt{N}$ 까지의 홀수로 완전히 나누어 지는가를 검사하는 명백한 검사방법은 거의 $\frac{1}{2}\sqrt{N}$ 나누기를 요구하는데 이것은 대략 $10^{d/2}$ 이다. 다른 측면에서 보면 이 문제는 *NP*완성시키는데는 충분하지 못하다. 따라서 이것은 본질을 따르지 못한 문제로 된다. 여기서는 그 복잡한 내용을 언급하지 않는다.

여기서는 기본적인것을 검사할수 있는 다항시간알고리듬을 줄것이다. 가령 알고리듬이 그 수가 씨수가 아니라고 선언하면 물론 그 수는 씨수가 아니다. 가령 알고리듬이 그 수가 씨수라고 정의하면 높은 확률에서(100%는 아닌) 그 수는 확실히 씨수이라는것이다. 나머지확률은 오유확률로서 이것은 검사되는 개별적인 수에 관계되는것이 아니라 알고리듬에 의해 생성되는 란수의 선택에 관계된다. 그러므로 이 알고리듬은 아직 좀 미숙한데는 있으나 오유확률을 대수롭지 않게 보아도 될것이다.

알고리듬의 중심은 잘 알려 진 정리 즉 페르마(Fermat)정리이다.

정리 10-10.

페르마의 소정리(*Fermat's Lesser Theorem*); *P*가 씨수이고 $0<A<P$이면 $A^{P-1}\equiv 1 (\bmod P)$ 이다.

증명:

이 정리는 수론에 관한 임의의 책에서 증명되고 있다.

실례로 67이 씨수라면 $2^{66}\equiv 1 (\bmod 67)$이다. 이것은 수 *N*이 씨수인지 아닌지를 검사하는 알고리듬을 제기한다. $2^{N-1}\equiv 1 (\bmod N)$인가 하는것은 간단히 검사할수 있다. 가령 $2^{N-1}\neq 1 (\bmod N)$이라면 *N*은 확실히 씨수가 아니라는것을 알수 있다. 다른 측면에서 만일 동일성이 성립한다면 *N*은 대체로 씨수이다. 실례로 $2^{N-1}\equiv 1 (\bmod N)$을 만족시키나 씨수가 아닌 제일 작은 수 *N*은 *N*=341이다.

이 알고리듬은 부차적이기는 하나 오유가 있고 문제로 되는것은 언제나 같은 오유를 발생시킨다는것이다. 또다른 방도를 보면 그것이 다른 조작을 하지 않도록 하는 고정된 모임 *N*이 있다. 아래와 같이 알고리듬을 란수화하도록 시도할수 있다. 즉 란수에서 $1<A<N-1$인 *A*와 *N*을 고른다. 만일 $A^{N-1}\equiv 1 (\bmod N)$이라고 하면 *N*이 대체로 씨수라는것을 선언하며 그렇지 않으면 *N*은 정확히 씨수가 아니라는것이다. 가령 *N*=341이고 A=3이면 $3^{340}\equiv 56 (\bmod 341)$을 구할수 있다. 그러므로 이 알고리듬은 *N* =3으로 선택하면 *N* =341에 대하여 정확한 대답을 얻을수 있을것이다. 비록 이것이 그럴듯해도 선택된 모든 A에 대하여 이 알고리듬이 잘못된 수를 만들어 낼수도 있다.

```
/**
 *   Function that implements the basic primality test.
 *   If witness does not return 1,    n    is definitely composite.
 *   Do this by computing a^i (mod n) and looking for
 *   nontriviat square roots of 1 along the way
 */
HugeInt witness( const Hugeint & a, const Hugeint & i, const Hugeint & n )
{
        if( i == 0 )
            return 1;
        Hugeint x = witness( a, 1 / 2. n );
        if( x == 0 )       // If n is recursively composite, stop
            return 0;
        // n is not prime if we find a nontnviat square root of 1
        Hugeint y = ( x * x ) % n;
        if( y ==1 && x != 1 && x != n – 1 )
                      return 0;
        if( i % 2 != 0 )
             y = ( a * y ) % n;
        return y;
 }
  /**.
   *   The number of witnesses queried in randomized primality test.
   */
  static const int TRIALS = 5;
  /**
   *   Randomized primality test.
  *   Adjust TRIALS to increase confidence level.
  *   n is the number to test.
  *   If return value is false, n is definitely not prime.
  *   If return value is true, n is probably prime.
  * bool 1s Prime( constHugeint & n )
  */

 {
          Random r;

          for( int counter = 0; counter < TRIALS; counter++ )
                    if( witness( r.randomInt( 2, (int) n-2),n-l,n)!=l)
                          return false;
          return true;
 }
```

프로그램 10-12. 확률적인 씨수성검사알고리듬(가상부호)

이제 **카마이클**(c*armichael*)이라고 하는 하나의 수를 설정하자. 이것들은 씨수는 아니지만 N에 대해 상대적으로 씨수인 모든 $0 < A < N$에 대하여 $A^{N-1} \equiv 1(\mathrm{mod}\ N)$이 성립한다. 이때 제일 작은 수는 561이다. 따라서 오유가 없도록 하기 위하여 추가적인 검사가 필요된다.

제7장에서 2차적인 증명에 관계되는 정리를 고찰하였다. 이 정리의 특별한 경우는 아래와 같다.

정리 10-11.

P가 씨수이고 $0<X<P$이면 $X^2\equiv1(\mathrm{Mod}\ P)$의 풀이는 오직 X=1, P-1뿐이다.

증명:

$X_2\equiv1(\mathrm{Mod}\ P)$는 $X_2-1\equiv0(\mathrm{Mod}\ P)$라는것을 의미한다. 이것은 또한 $(X-1)$ $(X=1)\equiv0(\mathrm{mod}$ $P)$라는것을 의미한다. 따라서 P가 $0<X<P$인 씨수이라는데로부터 P는 $(X$-1) 혹은 $(X$+1)로 나누어 져야 한다. 정리를 따르자.

그러므로 임의의 점에서 $A_{N\text{-}1}$ (mod N)의 계산에 대하여 이 정리와 어긋나게 쓴것을 밝혀 낸다면 정확하게 N은 씨수가 아니라는것을 결론할수 있다. 가령 제2장 제4절 4에서 power를 쓰면 이것을 시험하는데 아주 좋은 기회로 될것이다. N Mod연산을 수행하도록 이 방법을 수정하고 정리 10-11을 검사하는데 적용하게 된다. 이 전략은 프로그람 10-12에서 보여 준 가상부호로 수행된다.

만일 Witness가 임의의 값 즉 1을 준다면 그것은 N이 씨수일수 없다는 증명을 하게 된다. 이 증명은 인수를 실지로 찾아 내는 방법으로는 되지 못하기때문에 건설적이지는 못하다. 이것은 임의의 충분하게 큰 N에 대하여 기껏해서 A의 값이 (N-9)/4인 이 알고리듬에 의한것이라는것을 보여 주고 있다. 그러므로 A가 란수에서 선택되고 그리고 알고리듬이 N이 씨수라고 한다면 이 알고리듬은 적어도 75%는 정확하다. Witness가 50번 집행된다고 가정하자. 그러면 오동작하는 확률은 기껏해서 1/4이다. 그러므로 50회의 독자적인 란수시험에서 오차한계는 $1/4^{50}=2^{-100}$보다 더 크지 않다.

제5절. 역추적알고리듬

이 장에서 고찰하게 될 마지막알고리듬설계기술은 **역추적알고리듬**(*Backtracking Algorithms*)이다. 많은 경우 역추적알고리듬은 불리한 실행환경에서 재빠른 탐색을 재치있게 진행하도록 한다.이것은 언제나 그런것은 아니며 가장 철저히 탐색하여 보존해야하는 경우 등에서 중요한 알고리듬으로 간주된다. 물론 알고리듬집행은 다음의것에 관계된다. 즉 $O(N^2)$알고리듬은 정렬을 위해서는 그렇게 좋은 편이 못되나 $O(N^5)$알고리듬은 순회판매(혹은 임의의 NP완전)문제에 대한 목적하는 결과를 줄것이다.

역추적알고리듬의 실천적인 실례는 새 집에서 가구를 배치하는 문제이다. 문제를 풀자면 많은 가능성이 있으나 다만 일반적으로 몇개만이 고려된다. 배렬되지 않은 상태에

서 매개 가구는 방의 아무 위치에 놓는다. 만일 모든 가구를 놓고 만족하다면 알고리듬은 끝난다. 가령 가구를 놓은 위치가 맞춤하지 않아 위치가 마음에 없다면 다른것과 교체해야 한다. 물론 이 단계에서 또 다른 배치안을 취소하고 앞 단계로 넘어 올수 있다. 만일 모든 가능한 첫 단계까지 취소하게 된다면 더 훌륭한 가구배치안은 없다. 그렇지 않은 경우 가장 충분한 배치를 하고 작업을 끝낸다. 언급할것은 이 알고리듬이 가능한 모든 가능성을 즉시에 취소할수도 있다는것이다. 실례로 부엌에 쏘파를 두는것을 고려하는 배렬은 아예 시도조차 하지 않는다. 대다수 잘못된 배치는 쉽게 포기할수 있으며 이로하여 배치에서 달갑지 않은 부분모임은 제거한다. 현 단계에서 가능한 큰 범위를 제거하는것을 **자르기**(*pruning*)라고 한다.

2개의 역추적알고리듬실례를 여기서 보게 된다. 첫 문제는 **계산기하학문제**(*computational geometry*)이며 두번째 문제는 **서양장기**(*chess*)나 **살창무늬서양장기**(Cheeker)와 같은 유희에서 콤퓨터대상이 어떻게 움직이는가를 보여 주는 문제이다.

1. 통행료금소재구축문제

*N*개의 점 $p_1, p_2 \cdots p_N$이 x축에 놓여 있다고 가정하자. x_i는 P_i의 x자리표점이다. x_1=0이고 점들은 왼쪽에서 오른쪽으로 주어 진다고 하자. 이 *N*개의 점들로 $N(N-1)/2$ (반드시 일치하지 않음)개의 거리를 계산할수 있는데 거리 d_1, $d_2 \cdots d_N$은 매개 점쌍들에 대하여 $|x_i-x_j|$ $(i \neq j)$형태의 점들사이의 거리이다. 점들을 정하게 되면 시간계산량 $O(N^2)$으로 거리들을 계산하는것은 힘들지 않다. 이렇게 하는것은 물론 정렬은 되지 않았으나 계산시간한계를 $O(N^2\log N)$로 정돈해 놓으면 거리들은 정렬될수 있다. **통행료금소재구축문제**(*turnpike reconstruction problem*)는 그 거리로부터 점설정을 다시 하는 문제이다. 이 문제는 물리와 분자생물학(좀더 특이한 정보자료를 위하여 참고서를 보시오.)에서 많이 응용되는 문제이다. 이 명명(이름짓기)은 점들이 동쪽의 높은 변두리에서부터 되돌아서 빠져 나간다는 분석으로부터 지어 진것이다. 재구축문제는 구축문제보다는 좀 어려워 질것으로 보아 진다. 현재까지 이 알고리듬이 다차원시간에 실행된다고 담보를 줄수 있는 제안은 하나도 없다. 이 알고리듬은 일반적으로 $O(N^2\log N)$시간에 동작할수 있으나 최악의 경우에는 지수함수시간량만큼 걸릴수 있다.

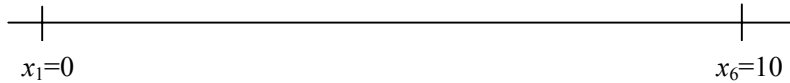
물론 이 문제에서 한가지 풀이가 주어 지면 그것들의 다른 미지수는 모든 점들에 편의점을 더해 줌으로써 구축될수 있다. 이 문제에서 첫 점을 0으로 정하며 풀이는 커지는 순서대로 출력되도록 정한다.

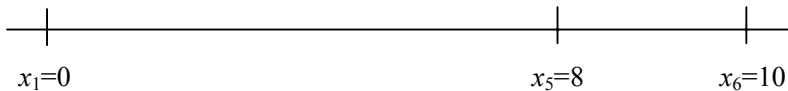
*D*는 거리들에 대한 설정값이고 $|D|=M=N(N-1)/2$이라고 하자.

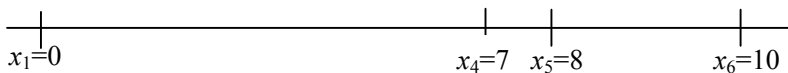
$$D = \{1, 2, 2, 2, 3, 3, 3, 4, 5, 5, 5, 6, 7, 8, 10\}$$

여기서 $|D|$=15이므로 *N*=6이라는것을 알수 있다. 먼저 X_1=0으로 설정하고 알고리듬의

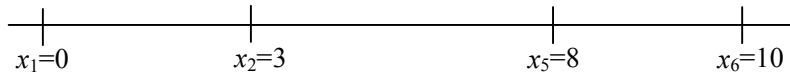
풀이를 시작한다.

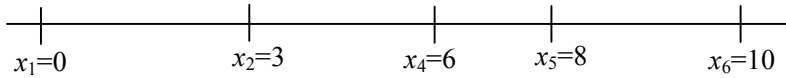
명백히 제일 큰 원소는 10이므로 X_6=10이다. D에서 10을 제거한다. 이때 배치된 점들과 그것들사이의 거리들을 아래에 보여 준다.

D = {1, 2, 2, 2, 3, 3, 3, 4, 5, 5, 5, 6, 7, 8}

나머지거리에서 제일 큰 값은 8인데 그러면 X_2=2 혹은 X_5=8이라는것을 의미한다. 균형적으로 볼 때 그 선택이 중요치 않다는것을 결론 지을수 있으며 이로부터 2개중 풀이에 영향을 주도록 선택을 하든지 아니면 그렇게 하지 않을수도 있다. 그래서 풀이에 영향을 주지 않는 X_5=8을 선정할수 있다. 다음 D에서 거리 x_6-x_5=2와 x_5-x_1=8을 제거한다.

D= {1, 2, 2, 3, 3, 3, 4, 5, 5, 5, 6, 7}

다음 단계는 명백하지 않다. D에서 제일 큰 거리가 7이므로 x_4=7이든지 x_2=3이다. 만일 x_4=7이라면 거리 x_6-7=3이며 x_5-7=1가 D에 포함되여 있어야 한다. 재빨리 속셈해 보면 이것들은 옳다는것을 알수 있다. 다른 한편 x_2=3으로 보면 3-x_1=3이고 x_5-3=5이 D에 포함되여 있어야 한다. 이 거리들은 물론 D에 포함되여 있으며 이로부터 선택이 문제로 되지 않는다는것을 알수 있다. 그래서 두번째 시도는 버리고 첫번째 시도에로 되돌아 가서 선택한 x_4=7을 풀이에 적용하게 된다. 그러면 x_4=7의 확정은 거리 7을 D에서 제거한다.

D = { 2, 2, 3, 3, 4, 5, 5 , 6, 7}

이 점에서 x_1=0, x_4=7, x_5=8, x_6=10을 얻게 된다. 이제 더 긴 거리는 6이므로 x_3=6 혹은 x_2=4로 취한다. 그런데 x_3=6이면 x_4-x_3=1이며 이것은 가능한 방안이 못된다. 그것은 1이 거리모임 D에서 보다 긴것이 못되기때문이다. 다른 한편 x_2=4이면 x_2-x_0=4, x_5-x_2=4이다. 이것 또한 거리모임 D에서 한번만 출현하는 거리이기때문에 적합치 않다. 그래서 이 점들은 풀이로 되지 못하므로 다시 앞공정으로 되돌아 간다.

x_4=7이므로 풀이를 얻는것은 실패하였다. x_2=3으로 놓고 다시 시도한다. 이것역시 실패하면 풀이가 없음을 통지한다. 현재 다음의 상태를 가진다.

D={1, 2, 2, 3, 3, 4, 5, 5, 6}

다시한번 x_4=6, x_3=4로 선택해야 한다. x_3=4는 D모임에 거리 4가 한번만 출현하기때문에 불가능하며 이 두개 점은 이 선택에서 적합치 못하다. X_4=6은 가능하므로 포함시킨다.

D={1, 2, 3, 5, 5}

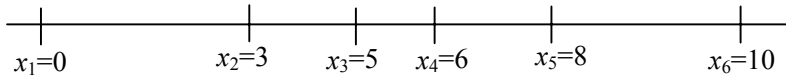
한가지 남은 선택은 x_3=5라고 하는것인데 이것은 D모임을 빈 모임으로 만들며 따라서 하나의 풀이를 가진다.

D={}

그림 10-42는 풀이에 도달하기 위하여 취해야 할 상태들을 표시하는 결정나무구조를 보여 준다.

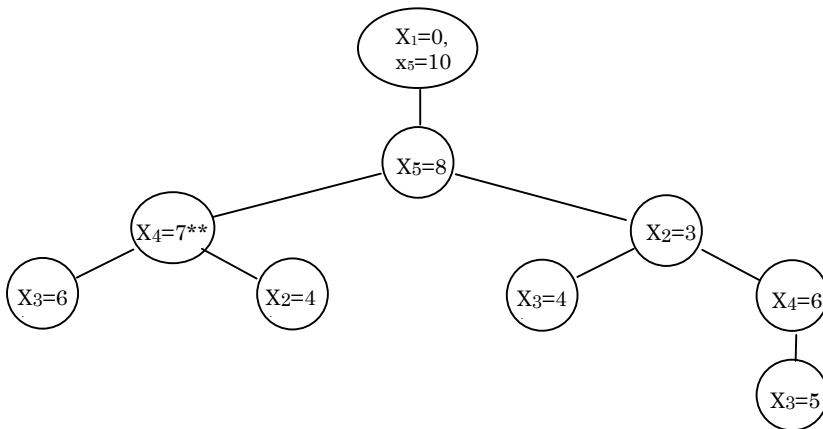



그림 10-42. 통행료금소재구축실례로 만들어 진 결정나무

가지들을 표시하는것대신에 우의 그림과 같이 가지의 목적매듭에 표식한다. 별표식을 가진 매듭은 선택된 점들이 주어 진 거리와 일치되지 않는것을 표시하며 두개의 별표식을 한 **매듭표시**는 자식매듭으로서는 불가능한 매듭들을 가지고 있음을 표시하며 이로

부터 정확치 못한 경로를 표시한다.

```
           bool turnpike( vector<int> & x, DistSet d, int n )
           {
/*1*/          x[ 1 ] = 0;
/*2*/          d.de1eteMax( x[ n ] );
/*3*/          d.de1eteMax( x[ n - 1 ] );
/*4*/          if( x[ n ] - x[ n - 1 ] ∈  d )
               {
 /*5*/             d. remove ( x[n]-x[n-l]);
 /*6*/             return place( x, d, n, 2, n - 2 );
               }
               else
 /*7*/         return false;
           }
```

프로그람 10-13. 통행료금소재구축알고리듬
구동프로그람(가상부호)

이 알고리듬을 실현하는 가상부호는 더할나위없이 정확하다. 구동루틴 turnpike는 프로그람 10-13에 보여 준다. 이 알고리듬은 배렬점 *x*(이 변수는 초기화되지 말아야 한다.)와 거리모임 *D*와 *N*을 인수로 받는다. 만일 풀이가 있다면 true를 되돌려 줄것이며 *x*에 그 결과값들이 배치되고 *D*는 빈 모임이 될것이다. 풀이가 없다면 false가 얻어 지며 *x*는 정의되지 않고 거리모임 *D*는 빈 모임이 아닐것이다. 그 알고리듬은 우에서 본것처럼 X_1, X_{N-1}, X_N을 설정하며 *D*를 변경시키고 다른 점들을 배치하기 위하여 역추적알고리듬 place를 호출한다. $|D|=N(N-1)2$이 성립하는가를 이미 검사하였다.

역추적알고리듬이 좀 어려운 알고리듬이라는것은 프로그람 10-14에서 알수 있다.

```
            /**
            * Backtracking algorithm to place the points x[left] ... x[right].
            * x[l].. ,x[left-l] and x[right+l]..,x[n] already tentatively placed .
            * If place returns true, then x[left].. .x[right] will have values.
            */
            bool p1ace( vector<int> & x, DistSet d, int n, int left, int right )
            {
               int dmax;
               bool found = false;
/* 1 */       if( d.isEmpty( ) )
/* 2 */            return true;
/* 3 */         dmax = d.findMax( );
            // Check if setting x[right] = dmax is feasible.
```

```
/* 4 */          if( | x[j] - dmax |∈   d for a11 l≤j<left and right<j≤n )
                 {
/* 5 */            x[right] = dmax;                       // Try x[right]=dmax
/* 6 */              for( l≤j<1eft, nght<j<n )
/* 7 */                 d. remove (    | x[j] - dmax | );
/* 8 */              found = place( x, d, n, left, right-1 );
/* 9 */              if ( !found )            // Backtrack
/* 10 */                        for( 1 ≤j<left, right<j≤n ) // Undo the deletion
/* 11 */                           d.insert( | x[j] - dmax | );
                 }
                 // If first attempt failed, try to see if setting
                 // x[1eft]=x[n]-dmax is feasible.
/* 12 */             if( !found && ( | x[n] - dmax - x[j] |∈   d
/* 13 */                              for a11 1<j<1eft and right<j  ≤n ) )
                     {
/* 14 */                x[ left ] = x[n] - dmax;           // Same logic as before
/* 15 */                for( l  ≤j<1eft, right<j<n )
/* 16 */                   d. remove ( | x[n] - dmax - x[j] | );
/* 17 */                   found = p1ace( x, d, n, left+1, right );
/* 18 */              if( ! found )            // Backtrack
/* 19 */                   for( 1<j<1eft, right<j≤n ) // Undo the deletion
/* 20 */                       d.insert( | x[n] - dmax - x[j] | );
                     }
/* 21 */             return found;
               }
```

프로그램 10-14. 통행료금소재구축알고리듬:역추적단계들(가상부호)

대부분의 역추적알고리듬과 마찬가지로 가장 편리한 실현은 재귀이다. 여기에서는 같은 인수들을 경계 *Left*와 *Right*로 표시한다. 즉 $x_{Left} \cdots x_{Right}$는 배치하려는 점의 x자리표이다. 만일 D모임이 비였다면 (혹은 *Left*>*Right*) 풀이가 구해 지며 결과를 넘겨 줄수 있다. 그렇지 않으면 먼저 $x_{Right}=D_{max}$로 한다. 그에 합당한 거리모두가 존재하면(정확한 값으로) 이 점을 시험적으로 정하고 이 거리를 제거하며 *Left*에서 *Right*-1만큼 옮긴다. 만일 그런 거리가 없거나 *Left*로부터 *Right*-1에로의 옮기기가 실패하면 같은 방법으로 $x_{Left}=x_N-d_{max}$로 설정해야 한다. 만일 그렇게 할수 없다면 풀이는 없으며 그렇게 할수 있다면 풀이는 존재한다. 그리고 이 정보는 결국 return명령에 의하여 turnpike에 x배렬을 되돌려 준다.

이 알고리듬의 분석은 두가지 사실을 시사해 준다. 9행부터 11행, 18행부터 20행이 실행되지 않는다. D는 균형탐색(혹은 splay)나무(이것은 물론 부호수정을 요구한다.)와 같이 설명할수 있다. 만일 역추적을 절대로 안한다면 D가 실행되는 연산시간량은 최대한 $O(N^2)$이며 지우기와 탐색은 4행과 12부터 13행까지에서 진행된다. 이것은 D가 $O(N^2)$요소

들을 가지며 재삽입되는 원소는 없기때문에 지우기에 대하여서는 명백하다. Place에 대한 매개 호출은 최대한 $2N$회의 find를 리용하며 이 분석에서 place가 역추적을 하지 않기때문에 최대한 $2N^2$의 find연산들이 있을수 있다. 때문에 역추적을 하지 않으면 실행시간은 $O(N^2 \log N)$이다.

물론 역추적이 여러번 반복적으로 진행된다면 알고리듬의 실행에 영향을 미치게 된다. 이것은 비정상적인 경우의 자료구축으로 발생할수 있다. 시험결과는 만일 점들이 옹근자리표축에 고르게 표시되고 $[0, D_{max}]$에서 자유로운 값을 가진다면 전체 알고리듬을 처리하는 동안에 적어도 한번의 역추적이 진행된다는 명백한 결론을 할수 있다. 여기서 $D_{max} = O(N^2)$이다.

2. 유희

마지막응용으로서 콤퓨터에서 전략적인 **유희**(*game*) 즉 살창무늬**서양장기**(checker), 서양장기 취스와 같은것을 할수 있는 전략을 고찰하려고 한다. 실례로 세목놓기(tic-tac-toe)라고 하는 유희를 고찰하려고 하는데 그것은 이 유희가 설명하기 더 쉬운 문제점들을 가지고 있기때문이다.

세목놓기유희는 가령 두면들을 최적으로 오고 가면서 놀수 있다면 하나의 장기유희이다. 하나하나 주의깊게 분석해 보면 기회가 좋으면 언제나 승리하고 그렇지 못하기도 하는 이 알고리듬을 만드는것은 그리 어렵지 않다. 확실한 위치는 조종간(trap)으로 알게 되여 있고 또 참고표에 의하여 조종되기때문에 이것은 가능하다. 다른 유희전략은 그것이 가능한 때에 가운데4각형을 취하게 되는데 이것은 그 분석을 더 간단하게한다. 이것을 수행된면 표를 써서 오직 현재위치에 기초하여 이동해 갈수 있다. 물론 이 전략은 콤퓨터가 아니라 가장 지능처리능력이 높은 프로그람작성수가 짠다.

최대최소전략

더 일반적인 전략은 위치의 상대값을 나타내는 평가함수를 리용하는것이다. 콤퓨터가 이기게 되는 위치는 +1값을 가질 때인데 0을 가지면 비기게 되고 –1을 가지면 실패한다. 이 위치값은 경계위치라고 하는 판을 조사하여 결정할수 있다.

이 위치는 경계가 아니라면 그 위치값은 두개의 면을 최적으로 순환하면서 재귀적으로 결정된다. 이것은 한 선수(사람)는 다른 선수(콤퓨터)가 최대위치로 가려고 할 때 최소위치값으로 가려고 시도하는것으로 하여 일명 최대최소전략이라고 불리운다.

다음수위치(*successor position*) p는 p로부터 한번 이동하여 도달할수 있는 어떠한 위치 p_3이다. 가령 콤퓨터가 어떤 위치 p에서 이동하면 이것은 모든 다음수위치들의 값을

재귀적으로 평가한다. 콤퓨터는 먼저 가장 큰 값을 가지고 이동하는데 이 값은 p의 값이다. 임의의 다음수의 위치 p_s와 p_s의 다음수 모두를 재귀적으로 평가하고 가장 작은 값을 선택한다. 이 가장 작은 값은 사람선수에게 가장 적합한 응답을 준다.

프로그람 10-15의 부호는 콤퓨터의 가장 명확한 전략을 세운것이다. 1행에서 4행까지에서는 이기느냐 조작중이냐하는것을 즉시에 평가한다. 이 경우들중에 아무 조건에도 해당되지 않으면 그 위치는 경계가 아니다.

```
/**
 * Recursive function to find best move for computer.
 * Returns the evaluation and sets bestMove, which
 * ranges from 1 to 9 and indicates the best square to occupy.
 * Possible evaluations satisfy COMPJ-OSS < DRAW < COMP_WIN.
 * Complementary function findHumanMove is     Program 10-16.
*/
int TicTacToe::findCompMove( int & bestMove )
{
int i, responseValue;
int dc;     // dc means don't care; its value is unused
int value;
/*1*/        if( fullBoard( ) )
/*2*/              value = DRAW;
                 else
/*3*/        if( immediateCompWin( bestMove ) )
/*4*/            return COMP_WIN;       // bestMove wi1l be set by immediateCompWin
               else
               {
/*5*/              value = COMPJ-OSS; bestMove = 1;
/*6*/              for( i = 1; 1 <= 9; 1++ ) // Try each square
                     {
/*7*/                  if( isEmpty( i ) )
                                                  {
/*8*/                          place( i, COMP );
/*9*/                          responseValue = findHumanMove( dc );
/*10*/                         unplace( i ); // Restore board
/*11*/                         if( responseValue > value )
                                   {
                                     // Update best move
/*12*/                               value = responseValue;
/*13*/                               bestMove = i;
                                   }
                               }
                       }
                 }
/*14*/         return value;
           }
```

프로그람 10-15. 최대최소 세목놓기알고리듬: 콤퓨터가 먼저시작

value는 가능한 모든 다음수의 위치의 최대값을 포함할것이라는것을 다시 상기하자. 5행은 그것을 가능한 제일 작은 값으로 초기화하며 6행~13행의 순환은 보다 최적을 위한 탐색을 진행한다. 매 다음수의 위치는 8행부터 10행사이에서 반복적으로 평가된다.

이 과정은 반복적이며 여기에서 보게 되겠지만 findHumanmove함수는 findcompMove함수를 호출한다. 만일 사람이 콤퓨터가 이미 움직여 간것보다 더 적합한 위치로 움직여 가도록 응답한다면 Value와 best Move는 갱신된다. 프로그람 10-16은 사람의 움직임을 조종하도록 하기 위한 기능을 보여 준다. 이 론리는 선수가 가장 낮은 값을 가진 위치로 가도록 이동을 조종하는것을 제외하고는 같다. 사실 이 두개의 공정을 그것이 이동하도록 지시하는 외부변수를 통하여 하나로 결합하는것은 어렵지 않다. 이렇게 프로그람이 알아 보기 힘들게 작성되게 되므로 따로따로 분류하여 부호를 서술한다.

```
                int TicTacToe: :findHumanMove( int & bestMove )
                {
                int 1, responseValue;
                int dc;     // dc means don't care; its value is unused
                int value;
/*1*/                  if( fu11Board( ) )
/* 2*/                       value = DRAW;
                       else
/*3*/           if( immediateHumanWin( bestMove ) )
/*4*/                   return COMP_LOSS;
                    else
                    {
/*5*/               value = COMP_WIN; bestMove = 1;
/*6*/               for( i = 1; i <= 9; i++ ) // Try each square
                        {
/*7*/                   if( isEmpty( i ) )
                            {
/*8*/                       place( i, HUMAN );
/*9*/                       responseValue = findCompMove( dc );
/*10*/                     unplace( i ); // Restore board
/*11*/                     if( responseValue < value )
                                {
                                        // Update best move
/*12*/                              value = responseValue;
/*13*/                              bestMove = i;
                                 }
                            }
                        }
                    }
/*14*/              return value;
               }
```

프로그람 10-16. 최대최소세목놓기알고리듬;사람이 선택

련습문제를 통하여 이런 기능을 지원하는 프로그람들을 보도록 한다. 가장 품이 드는 계산은 콤퓨터가 조작중의 이동을 선택하려고 하는 경우이다. 이 단계에서 유희는 강제적으로 그리기를 실행하며 콤퓨터는 4각형[30]을 선택한다. 총체적으로 97162개의 위치들이 계산되였고 이 계산은 몇초밖에 걸리지 않는다. 부호를 최적화하는 시도는 하지 않는다. 콤퓨터가 두번째로 움직여 갈 때 계산되는 위치들의 수는 사람이 가운데 4각형을 선택하였다면 5,185이며 구석에 있는 4각형이 선택되였다면 9,761, 구석이 아닌 변두리의 4각형이 선택되였다면 13,233이다.

살창무늬서양장기와 취스와 같이 더 복잡한 유희들에서 경계매듭으로 가는 모든 길을 탐색하는것은 도저히 불가능하다. 정확한 재귀깊이에 도착한후 탐색을 정지해야 한다. 재귀가 정지된 매듭들은 경계매듭으로 된다. 이 경계매듭들은 위치값을 평가하는 함수에 의하여 평가된다. 실례로 서양장기프로그람에서 평가함수는 장기쪼각의 상대량과 길이포텐샬인수 등과 같은 변수들을 측정한다. 평가함수는 성공을 위해서 매우 중요한 함수이다. 그것은 콤퓨터의 이동선택이 이 함수의 최량화기능에 관계되기때문이다.

그러나 콤퓨터장기에서 제일 중요한 인자는 프로그람이 가능한 앞보기인자이다. 이 수는 때때로 *ply*라고 하며 반복하는 매듭의 깊이와 같다. 이것을 실현하기 위하여 외부파라메터가 탐색프로그람에 주어 진다.

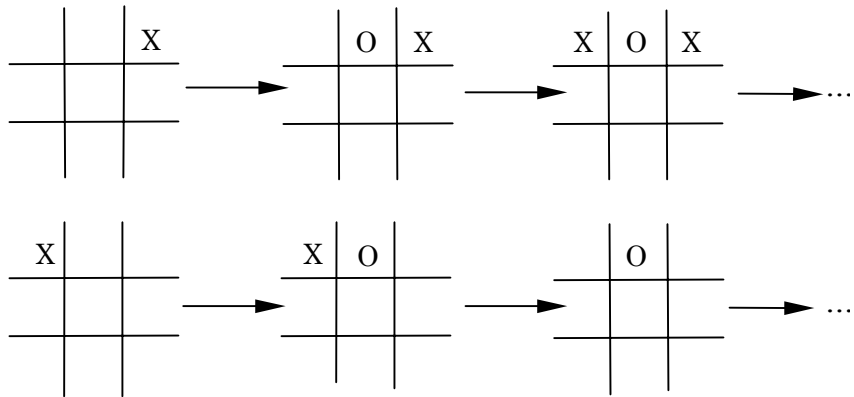



그림 10-43. 같은 위치에 도달하는 두 탐색들

유희프로그람에서 앞보기인수를 증가시키는 기본방법은 어떤 정보도 잃지 않는 몇개의 매듭을 방법으로 개선한다. 한가지 방법은 이미 보아 온것처럼 평가되여 온 모든 위치의 자리길을 보존하는데 표를 리용하는것이다. 실례로 첫 이동에 대한 탐색통로에서 프로그람은 그의 위치들을 검사해 볼것이다(그림 10-43에서). 가령 위치값이 기억(보존)되고 다음 위치의 출현이 다시 고려되지 않으면 본질적으로 이것은 끝위치로 되였음을

[30] 이 수는 통이 왼쪽우에서 시작하여 오른쪽으로 움직여 가는것을 세는 수이며 우의 루틴에 대해서만 유효하다.

말해 준다. 이것을 기록하는 자료구조를 변환표라고 하는데 이것은 언제나 하쉬표에 의하여 실행된다. 많은 경우에 이 표는 고려할수 있는 계산량을 줄일수 있다.

실례로 비교적 작은 장기알수를 가진 서양장기의 마감유희에서 탐색회수측정은 여러 준위 더 깊이 가도록 하는데 한번의 탐색을 진행하여 그 시간을 줄일수 있다.

$\alpha-\beta$ 가지자르기

가장 가치 있는 알고리듬중의 하나가 **$\alpha-\beta$ 가지자르기**($\alpha-\beta$ *pruning*)알고리듬이라고 할수 있다. 그림 10-44는 가상적인 유희에서 어떠한 가상적인 상태를 평가하는데 쓰이는 재귀적인 호출들의 결과를 보여 준다. 이것은 일반적으로 경기나무와 같은데서 참고된다(이것은 어느정도 유도되지 않는 즉 알고리듬에 의하여 실제나무가 구축되지 않기때문에 이때까지는 이 항목외 리용을 피하여 왔다. 이 경기나무는 이제 절대적인 개념으로 쓰일것이다.). 이 경기나무의 값은 44이다.

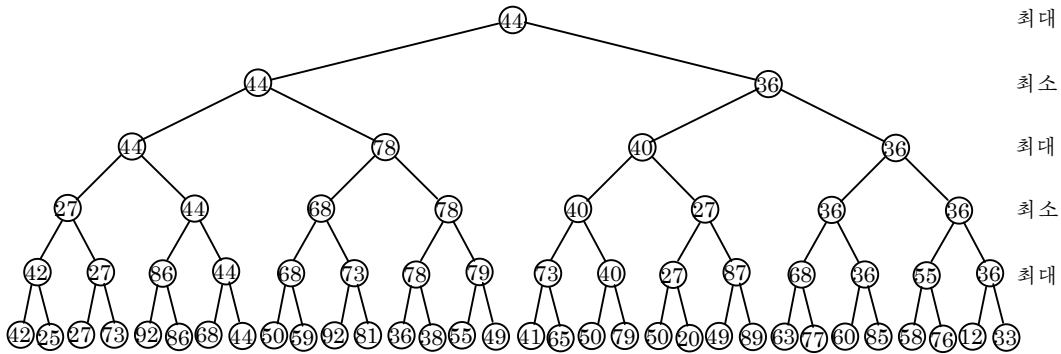



그림 10-44. 가상유희나무

그림 10-45에서는 여러개의 평가되지 않은 매듭을 가지고 우와 같은 경기나무를 평가한다. 경계매듭의 거의 절반은 검사된것이 아니다. 그 매듭들의 평가는 뿌리매듭의 값을 변화시키지 않는다는것을 보여 준다.

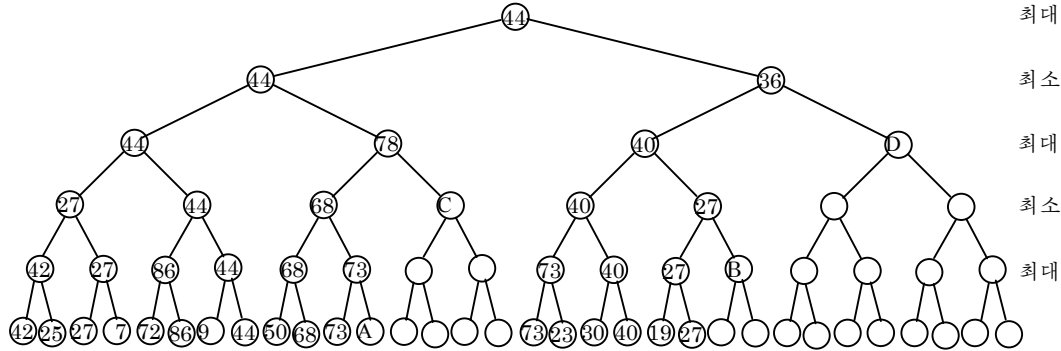



그림 10-45. 잘리워 진 유희나무

먼저 매듭 *D*를 고찰하자. 그림 10-46는 *D*를 평가할 때 얻어 지게 되는 정보를 보여준다. 이 시점에서 여전히 findHumanMove상에 있으며 *D*에 대한 FindComp Move에로의 호출을 주시하고 있다.

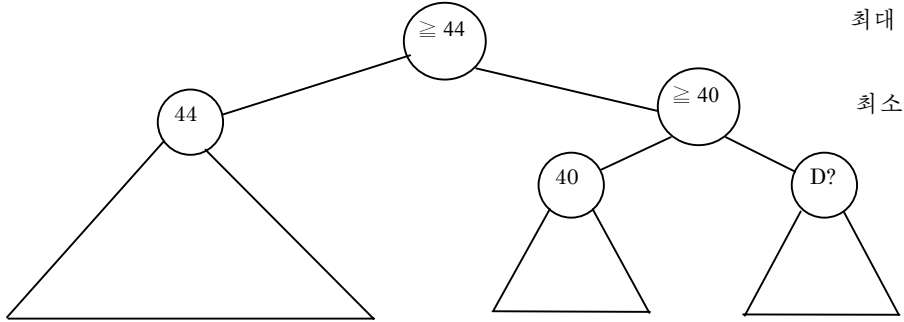



그림 10-46. ? 가 표시된 매듭은 중요하지 않다.

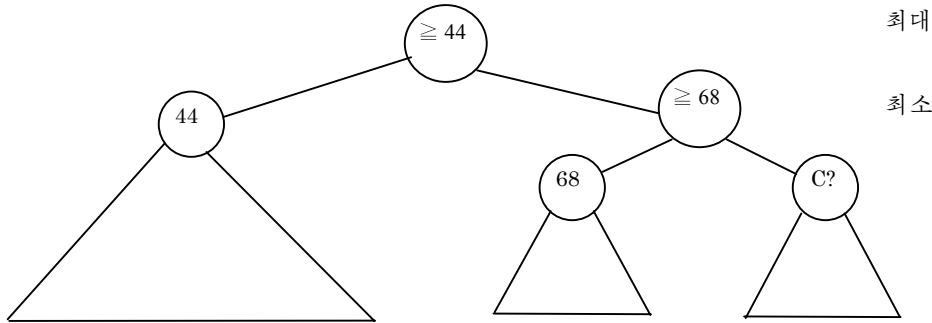



그림 10-47. ?가 표시된 매듭은 중요치 않다.

어쨌든 findHumanMove는 기껐해서 40을 되돌려 줄것이라는것을 이미 알고 있으며 이로부터 이것은 min매듭이다. 다른 한편 이 max매듭의 부모매듭은 값 44로서 이미 구해졌다. *D*는 결코 이 값을 증가시키지는 못한다. 그러므로 *D*는 평가가 필요없게 된다. 이 나무의 가지자르기는 α가지자르기로 알려 져 있다. 매듭 *B*에서도 같은 상태가 발생한다. α가지자르기를 실현하기 위하여 findCompMove는 이 최대시험값(α)을 findHumanMove로 준다. 만일 findHumanMove의 최대시험값이 이 값아래로 떨어 지면 findHuman Move는 곧 되돌이 된다.

류사하게 매듭 *A*와 *C*도 고찰된다. 이때 findCompMove 의 중간에서 *C*를 평가하기 위하여 findHumanMove를 호출한다. 그림 10-48은 매듭에서 충돌되는 상태를 보여 준다. 그러나 findHumanMove는 findCompMove가 호출되는 최소준위에서 기껏해서 44의 값을 택

할수 있도록 정한다(낮은 값들은 사람켠에서는 좋다는것을 다시 상기시킨다.). findcompMove가 최대시험값 68을 가지므로 그 무엇도 C가 min준위에서의 결과에 영향을 주지 않는다. 그러므로 C는 평가되지 않을것이다. 이 가지자르기형태는 β가지자르기로 알려 져 있으며 이것은 α가지자르기의 대칭적인 변형이다. 두 기술이 결합되면 **$\alpha-\beta$가지자르기**라고 할수 있다.

$\alpha-\beta$가지자르기의 실행은 놀랍게도 적은 행의 프로그람을 요구한다. 프로그람 10-17은 $\alpha-\beta$가지자르기전략의 절반을 보여 준다. 즉 다른 절반의 부호작성품을 덜어 줄수 있다.

$\alpha-\beta$가지자르기의 완전한 개선을 위하여 유희프로그람을 항상 가장 좋은 탐색위치로 쉽게 움직이도록 시도하는데서 말단이 아닌 중간매듭들에 대하여 평가함수를 쓰도록 한다. 이 결과 매듭들을 자유로 순서화하여 얻어 지게 되는 가지자르기보다는 더 좋은 가지자르기가 얻어 진다. 다른 기술은 고찰하게 되는 실제행에서 보다 더 깊게 탐색을 진행하는 기술인데 이것 역시 품이 더 든다.

실천에서 $\alpha-\beta$가지자르기는 탐색매듭들을 오직 $O(\sqrt{N})$개로 제한한다. 여기서 N은 옹근유희나무의 크기이다. 이것은 대단한 소득이며 $\alpha-\beta$가지자르기를 써서 진행하는 탐색은 가지자르기하지 않은 나무와 비교해 볼 때 2배나 더 깊게 탐색해 들어 갈수 있음을 의미한다. 세목놓기실례는 초기탐색매듭수 97162개로부터 4,493개의 매듭(이 수들은 경계가 아닌 매듭들을 포함하는 수)까지 줄일수는 있으나 동일한 값들이 굉장히 많아서 그렇게 리상적이지는 못하다.

많은 유희들에서 콤퓨터는 세계적으로 가장 우수한 선수들을 따라 간다. 여기에 리용된 기술들은 대단히 흥미 있으며 그이상의 여러가지 문제들에 적용될수 있다. 보다 구체적인 설명은 참고서에 가서 볼수 있다.

```
    * Same as before, but perform alpha-beta pruning.
    * The main routine should make the call with
    * alpha = COMP_LOSS and beta = COMP_WIN.
    */
    int TicTacToe: :findCompMove( int & bestMove, int alpha, int beta )
    {
    int i, responseValue;
    int dc;      // dc means don't care; its value is unused
    int value;
/*1*/          if( fu11Board( ) )
/*2*/             value = DRAW;
               else
/*3*/          if( immediateCompWin( bestMove ) )
/*4*/        return COMP_WIN;       // bestMove will be set by immediateCompWin
               else
               {
```

```
/*5*/                  value = alpha; bestMove = 1;
/*6*/                  for( i = 1; i <= 9 && value < beta; i++ ) // Try each square
{
/*7*/                      if( isEmpty( i ) )
              {
/* 8 */                  p1ace( i, COMP );
/* 9 */                  responseValue = findHumanMove(dc );
/*10 */                  unp1ace( i ); // Restore board
/*11*/                  if( responseValue > value )
                   {
                              // Update best move
/*12*/                        value = responseValue;
/*13*/                        bestMove = i;
                      }
                   }
                }
              }
/*14*/      return value;
            }
```

프로그램 10-17. α−β가지자르기에 의한 최대최소세목놓기
(세목놓기)알고리듬:콤퓨터선택.

요약

이 장에서는 알고리듬설계에서 리용되는 다섯가지의 가장 보편적인 기술들을 설명하였다. 어떤 문제에 부닥쳤을 때는 이것들중 어느것을 적용하겠는가를 잘 따져 볼 필요가 있을것이다. 자료구조를 잘 설계하고 정확한 알고리듬을 선택한다면 풀이를 빨리, 효률적으로 얻어 낼수 있는것이다.

련습문제

10-1. 다중처리기일감일정을 작성하기 위한 평균수행시간을 최소화하는 탐욕알고리듬을 설명하시오.

10-2. 입력으로서 일감 $j_1, j_2 \cdots j_N$들의 모임이 있는데 매개 일감은 1개의 시간단위에 완수된다고 한다. 매개 일감 j_i는 시간한계 t_i에서 완수되면 d_i만한 돈을 벌지만, 이 시간한계이후에 완수되면 조금도 벌지 못한다고 한다.

ㄱ. 이 문제를 풀기 위한 $O(N2)$의 탐욕알고리듬을 설계하시오.

**ㄴ. 우의 알고리듬을 수정하여 $O(N\log N)$시간한계를 계산하시오. 요령:시간

한계는 일감들을 돈에 따라 정렬하는데 전적으로 기인된다. 알고리듬의 나머지부분은 격리된 모임자료구조를 리용하면 $O(N\log N)$동안에 완성될수 있다.

10-3. 어떤 파일에 두점, 공백, 새행기호, 반두점 그리고 수자들만이 다음의 출현빈도로 포함되여 있다고 한다; 두점(100), 공백(605), 새행기호(100), 반두점(705), 0(431), 1(242), 2(176), 3(59), 4(185), 5(250), 6(174), 7(199), 8(205), 9(217)하프만부호를 작성하시오.

10-4. 부호화된 파일의 부분은 반드시 하프만부호를 지적하는 머리부여야 한다. 기껏해서 $O(N)$(이외에 기호들이 추가됨)크기의 머리부를 구축하기 위한 방법을 결정하시오. 여기서 N은 기호들의 개수이다.

10-5. 하프만의 알고리듬이 최량의 앞배치부호를 발생시킨다는 증명을 완성하시오.

10-6. 기호들이 출현빈도에 의해 정렬되면 하프만의 알고리듬이 선형시간에 완수될수 있다는것을 설명하시오.

10-7. 하프만의 알고리듬을 리용하여 파일압축(그리고 풀기)을 진행하는 프로그람을 작성하시오.

***10-8.** 다음의 항목들의 렬이 있다. 즉 N개 항목들은 크기가 $1/6-2\varepsilon$, N개 항목들은 크기가 $\frac{1}{3}+\varepsilon$, N개 항목들은 크기가 $\frac{1}{2}+\varepsilon$이다. 그러면 임의의 직결상자채우기알고리듬이 적어도 3/2배의 최적상자수들을 리용하도록 할수 있다는것을 설명하시오.

10-9. $O(N\log N)$시간안에 처음적합과 최적적합을 수행하는 방법을 설명하시오.

10-10. 제10장 제1절 3에서 설명한 모든 상자채우기전략들의 다음의 입력렬우에서의 동작과정을 설명하시오;0.42, 0.25, 0.27, 0.07, 0.72, 0.86, 0.09, 0.44, 0.50, 0.68, 0.73, 0.31, 0.78, 0.17, 0.79, 0.37, 0.23, 0.30.

10-11. 여러가지 상자채우기경험규칙들의 성능(리용된 상자들의 수와 시간에 대한)을 비교하는 프로그람을 작성하시오.

10-12. 정리 10-7을 증명하시오.

10-13. 정리 10-8을 증명하시오.

***10-14.** N개의 점들이 단위4각형안에 놓여 있다. 가장 가까운 두점사이의 거리가 $O(N^{1/2})$임을 설명하시오.

***10-15.** 최단점알고리듬에서, 그 strip단위면적안의 점들의 평균개수는 $O(\sqrt{N})$임을 증명하시오. 요령:앞문제의 결과를 리용하시오.

10-16. 최단점알고리듬을 실현하는 프로그람을 작성하시오.

10-17. 세분할의 중간중간값전략을 리용하면 고속선택(quickselect)의 점근적인 실

행시간은 얼마인가?

10-18. 7분할의 중간중간값을 리용한 고속선택은 선형적이라는것을 설명하시오. 이 방법이 증명에서 왜 리용되지 않는가

10-19. 제7장에서 준 고속선택알고리듬을 실현하시오. 고속선택은 5분할의 중간중간값을 리용하며 표본화알고리듬은 제10장 제2절 3의 뒤에 주었다. 실행시간을 비교하시오.

10-20. 5분할의 중간중간값을 계산하는데 리용된 정보의 대부분을 잃는다. 이 정보를 좀 더 잘 리용해서 많은 회수의 비교들을 없애버릴수 있는 방법을 설명해 보시오.

*10-21. 제10장 제2절 3의 뒤에서 설명한 표본화알고리듬의 분석을 완성하시오. 그리고 δ와 s의 값들을 선택하는 방법을 설명하시오.

10-22. 재귀적인 곱하기알고리듬이 X=1234와 Y=4321이 있는 때 XY를 계산하는 방법을 설명하시오. 모든 재귀계산들을 포함시키시오.

10-23. 두개의 복소수 $X=a+bi$와 $Y=c+bi$를 세번의 곱하기만을 리용하여 곱하는 방법을 설명하시오.

10-24. ㄱ. $X_LY_R+X_RY_L=(X_L+X_R)(Y_L+Y_R)-X_LY_L-X_RY_R$ 설명하시오.

ㄴ. 이것은 N비트의 수들을 곱하기 위한 $O(N^{1.59})$의 알고리듬을 제기한다. 이 방법을 이 책에서 준 풀이와 비교하시오.

10-25. *ㄱ. 원래 크기의 대략 $\frac{1}{3}$만한 5개의 문제들을 풀어서 두개의 수들을 곱하는 방법을 설명하시오.

**ㄴ. 이 문제를 일반화하여 임의의 상수 $\varepsilon>0$에 대해 $O(N^{\varepsilon})$의 알고리듬을 작성하시오.

ㄷ. 우의 ㄴ에서의 알고리듬이 $O(N\log N)$보다 더 좋겠는가?

10-26. 스트라센의 알고리듬이 2×2행렬들의 곱하기에서 교환을 리용하지 않는데 왜 그런가.

10-27. 2개의 70×70행렬들이 143, 640번의 곱하기들을 리용하여 곱해 질수 있다. 이것을 리용하여 스트라센의 알고리듬에서 얻어 진 한계를 개선시킬수 있는 방법을 설명하시오.

10-28. $A_1A_2A_3A_4A_5A_6$을 계산하는 최적방법은 무엇인가? 여기서 행렬들의 차원수는: A_1:10×20, A_2: 20×1, A_3: 1×40, A_4: 40×5, A_5: 5×30, A_6: 30×15이다.

10-29. **련결목록행렬곱하기**를 위한 다음의 탐욕알고리듬들중 어느것도 처리되지 않는다는것을 설명하시오. 매 단계에서

ㄱ. 가장 시간이 덜 드는 곱하기를 계산한다.

ㄴ. 가장 시간이 많이 드는 곱하기를 계산한다.

ㄷ. 두 행렬 M_i과 M_{i+1}사이의 곱하기를 계산하는데 이때 M_i안의 렬의 개수는 최소이다(우의 규칙들중 어느 하나에 의해 중지된다.).

10-30. 행렬곱하기의 최적순서를 계산하는 프로그람을 작성하시오. 실제적인 순서를 인쇄하는 루틴을 포함시키시오.

10-31. 다음의 단어들에 대하여 최적2진탐색나무를 설명하시오. *a* (0.18), *and* (0.19), *I* (0.23), *it* (0.21), *or* (0.19)괄호안의 수는 출현빈도이다.

***10-32.** 비성공적인 탐색들에 허용되도록 최적2진탐색알고리듬을 확장하시오. 이 경우에 $1 \leqq j < N$에 대해 q_i는 $w_j < W < w_{j+1}$을 만족시키는 임의의 단어에 대해 탐색이 수행될 확률이다. q_0은 $W < w_1$에 대하여 탐색을 수행하는 확률이며 q_N은 $W < w_N$에 대해 탐색을 수행할 확률이다. 이때 $\sum_{i=1}^{N} P_i + \sum_{j=0}^{N} q_j = 1$임에 류의하시오.

***10-33.** $C_{i,j}$=0이며 한편 $C_{i,j}=W_{i,j}\min(C_{i,K-1}+C_{k,j})$라고 하자. W는 4각형부등식을 만족한다고 하자. 즉 모든 $i \leqq i' \leqq j \leqq ij'$ 에 대하여

$$W_{i,j}+W_{i',j'} \leqq W_{i,j}+W_{i',j'}$$

또한 W는 단조라고 가정하는바 즉 $i \leqq i'$ 이고 $j \leqq j'$ 이면 $W \leqq W_{i,j} \leqq W_{i',j'}$이다.

ㄱ. C가 4각형부등식을 만족한다는것을 증명하시오.

ㄴ. $R_{i,j}$를 최소의 $C_{i,k-1}+C_{k,j}$를 얻게 하는 가장 큰 k라고 하자(즉 동전들인 경우에 가장 큰 k를 고른다.).

다음 식이 성립함을 증명하시오.

$$R_{i,j} \leqq R_{i,j+1} \leqq R_{i+1,j+1}$$

ㄷ. R는 행과 렬에 따라 가면서 비감소임을 설명하시오.

ㄹ. 이것을 리용하여 C안의 매개 항목들이 $O(N^2)$시간으로 계산될수 있다는 것을 설명하시오.

ㅁ. 이 방법들을 리용하면 어떤 동적계획법알고리듬이 $O(N^2)$시간에 해결될수 있는가 ?

10-34. 제10장 제3절 4에 준 알고리듬으로부터 최단경로들을 재구축하는 루틴을 작성하시오.

10-35. 2항결수렬 $C(N,k)$가 다음과 같이 재귀적으로 정의될수 있다;

$C(N,o)=1$, $C(N,N)=1, 0 < k < N$에 대해

$C(N,k)= C(N-1,k)+ C(N-1,k-1)$

2항결수들을 다음과 같이 계산하기 위한 함수를 작성하고 실행시간을 분석하시오.

ㄱ. 재귀적으로 계산

ㄴ. 동적계획법을 리용하여 계산

10-36. 건너뛰기목록에서 삽입, 삭제, 탐색을 수행하는 루틴들을 작성하시오.

10-37. 건너뛰기목록연산들에 대해 연산시간이 $O(\log N)$이라는데 대해 형태상으로 증명하시오.

10-38. ㄱ. 콤퓨터상에서 란수발생기를 시험해 보시오. 란수특성이 어떠한가?

ㄴ. 프로그람 10-18은 돈을 튀기는 루틴을 보여 주는데 여기서 random은 옹근수를 되돌려 준다고 가정한다(이것은 많은 체계들에서 리용된다.).
란수발생기가 $M=2^B$형태의 모듈을 리용한다면(이것은 많은 체계들에서 리용되고 있다.)건너뛰기(skip)목록알고리듬의 성능은 어떻게 예견되겠는가?

10-39. ㄱ. 지수알고리듬을 리용하여 $2^{340}\equiv 1(\mathrm{mod}\ 341)$임을 증명하시오.

ㄴ. N=561에 대하여 A를 여러번 선택하여 우연적인 1차검사의 처리과정을 보여 주시오.

```
CoinSide flip( )
{
    if( ( random( ) % 2 ) == 0 )
        return HEADS;
    Else
        return TAILS;
}
```

프로그람 10-18. 애매한 동전튀기기문제

10-40. 통행료금소재구축알고리듬을 실현하시오.

10-41. 두개의 점모임들이 같은 거리모임을 가지며 서로 회전하지 않으면 그것들은 동일한 거리이다. 다음의 거리모임은 두개의 서로 다른 점모임들을 준다:{1, 2, 3, 4, 5, 6, 7, 8, 9 10, 11, 12, 13, 16, 17} 그 두개의 점모임들을 구하시오.

10-42. 재구축알고리듬을 확장하여 어떤 거리모임이 주어 졌을 때 모든 homometric점모임들을 구하시오.

10-43. 그림 10-48에 준 나무의 **$\alpha-\beta$ 가지자르기**의 결과를 설명하시오.

그림 10-48. 가지자르기 가능한 유희나무

10-44. ㄱ. 프로그람 10-17에 준 부호는 α가지자르기를 진행하는가 아니면 β가지자르기를 진행하는가?

ㄴ. 보충적인 루틴을 작성하시오.

10-45. 세목놓기놀이의 나머지과정을 작성하시오.

10-46. **1차원채우기문제**(*one dimentional circle packing*)는 다음과 같다: 반경이 $r_{1,}$ $r_{2,}\cdots r_N$인 N개의 원들이 있다. 이 원들을 어떤 4각형안에 채우되 매개 원은 4각형의 밑부분에 닿으며 원래의 순서대로 배렬되게 한다. 문제는 그러한 최소크기의 4각형의 너비를 구하는것이다. 그림 10-49에서 반경이 2, 1, 2 인 원들에 대한 하나의 실례를 보여 주고 있다. 가장 작은 4각형의 너비는 $4+4\sqrt{2}$ 이다.

그림 10-49. 원채우기문제의 실례

*10-47. 무방향그라프 G의 변들이 3각부등식 $c_{u,w,}\ c_{v,w,}\geq c_{u,w}$을 만족시킨다고 하자. 기껏해서 최적의 2배가 되는 순회판매경로를 계산하는 방법을 설명하시오.

요령 : **최소생성나무**(*minimum spanning tree*)를 구축하시오.

***10-48.** 경기지도원이 $N=2^k$선수들사이에서 련맹전경기를 조직할 필요가 있다고 한다. 이 경기에서 매 선수는 정확히 하루에 한번의 경기를 하며 N-1인 후에는 매 쌍의 선수들사이에서 한번의 경기가 있었다. 이것을 수행하는 재귀알고리듬을 작성하시오.

10-49. *ㄱ. 련맹전경기에서는 선수들을 $p_{i1}, p_{i2}, \cdots p_{iN}$의 순서로 배치하여 모든 $1 \leqq j < N$에 대하여 p_{i_j}가 $p_{i_{j+1}}$과의 경기에서 항상 이기도록 할수 있다는것을 증명하시오.

ㄴ. 그러한 한가지 배치형태를 탐색하는 $O(N\log N)$알고리듬을 구하시오. 그 알고리듬은 단락짓기문제(a)의 증명의 부분으로 될수도 있다.

***10-50.** 평면에 N개 점들의 모임 $p=p_1, p_2, \cdots p_N$이 주어 졌다. 보로노이선도는 이 평면을 N개의 구역 R_i로 분할한것인데 이때에 R_i안의 모든 점들은 P안의 임의의 다른 점보다도 p_i에 더 가깝도록 한다. 그림 10-50는 7개의 점(잘 배치된)들에 대한 **보로노이선도**(*voronoi diagram*)의 한가지 실례를 보여 주었다. 보로노이선도를 구축하는 $O(N\log N)$의 알고리듬을 구하여라.

***10-51.** 볼록다각형은 그 다각형우에 끝점들이 놓이는 임의의 선분이 그 다각형안에 완전히 놓이는 성질을 가지는 다각형이다. **볼록페포**(*convex hull*)는 평면에서 점들의 모임을 포함하는 면적이 최소인 볼록다각형을 구하는 문제이다. 그림 10-51은 40개의 점들의 모임에 대하여 볼록페포를 보여 주고 있다. 볼록**페포**를 탐색하는 $O(N\log N)$의 알고리듬을 구하시오.

***10-52.** 단락을 오른쪽 정돈하는 문제를 고찰하자. 단락은 길이가 $a_1, a_2, \cdots a_N$인 단어들 $w_1, w_2, \cdots w_N$의 렬을 포함한다. 이때 길이가 L인 행들로 끊으려고 한다. 단어들은 공백으로 분리되는데 공백의 리상적인 길이는 b(mm)이지만 필요한만큼 공백들이 연장되거나 혹은 압축되여(그러나 반드시 0보다 큰) 행 w_1, $w_{i+1} w_j$가 정확히 길이 L을 가지도록 할수 있다. 그런데 매개의 공백 b'에 대하여 $|b' - b|$개의 부조화점들을 축적하게 된다. 이에 례외로 되는것은 마지막행인데 이에 대해서는 $b' < b$인 때에만 축적한다(다시말하면 압축인 때에만 축적). 왜냐하면 마지막행은 조절될 필요가 없기때문이다. 그러므로 b_i가 a_i와 a_{i+1}사이의 공백의 길이라면 $j > i$인 임의의 행(마지막행은 제외) w_1, $w_{i+1} w_j$설정의 비조화성은 $\sum_{k=i}^{j-1} |b_k - b| = (j-i)|b' - b|$인데 여기서 b'는 이 행우의 공백의 평균크기이다. 이것은 $b' < b$인 때에만 마지막행에서도 성립하며 그렇지 않으면 마지막행은 언제나 비조화되지 않는다.

ㄱ. w_1, $w_{i+1}, \ldots w_n$을 가지고 길이가 L인 행들로 설정하는데서 최소비조화

설정을 구하는 동적계획알고리듬을 구하시오. 요령: $i=N$, $N-1$, … 1에 대하여 w_i, w_{i+1},… w_n을 설정하는 최선의 방도를 계산하시오.

ㄴ. 우에서 얻은 알고리듬에 대하여 시간과 공간의 복잡성(단어개수 N을 가지는 함수로서)을 구하시오.

ㄷ. 고정너비폰트를 리용하는 특정한 경우를 고찰하고 b의 최적값을 1(공백)이라고 가정하자. 이 경우에 공백에 대한 어떤 압축도 허용되지 않는다. 왜냐하면 다음번의 최소의 공백공간이 0으로 되기때문이다. 이 경우에 최소비조화설정을 발생시키는 선형시간알고리듬을 구하시오.

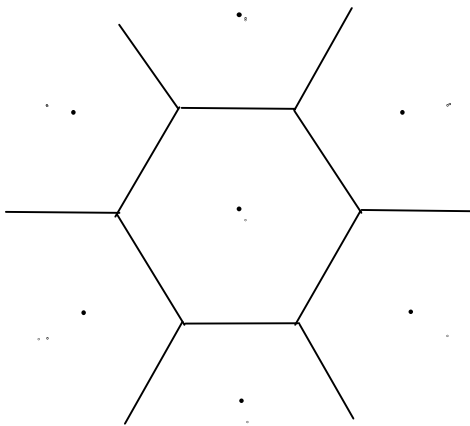
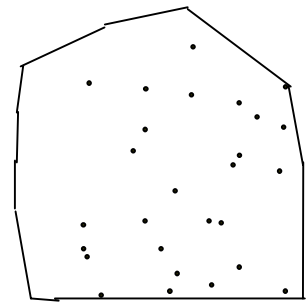



그림 10-50. 보로노이선도　　**그림 10-51.** **볼록페포**문제

*10-53. **최장증가부분렬**(*longest increasing subsequence*)문제는 다음과 같다:수 a_1, a_2,…, a_N들이 주어 지면 $a_{i1} < a_{i2} < \ldots a_{ik}$와 $i_1 < i_2 < \ldots i_k$를 만족시키는 k의 최대값을 구하여라. 례하면, 입력이 3, 1, 4, 1, 5, 9, 2, 6, 5이라면 최장증가부분렬은 길이가 4이다(그중의 하나는 1, 4, 5, 9). 최장증가부분렬문제를 푸는 $O(N^2)$알고리듬을 구하시오.

*10-54. **최장공통부분렬문제**(*longest common subsequence*)는 다음과 같다. 두개의 렬 $A=a_1\ a_2 \cdots a_M$과 $B=b_1,\ b_2\ \cdots,\ b_N$들이 주어 졌을 때 A와 B의 부분렬이면서 가장 긴 $C=c_1,\ c_2,\ \cdots,\ c_K$를 구하시오. 례하면 A=d, y, n, a, m, i, c이고 B=p, r, o, g, r, a, m, m, i, n, g 이라면 최장공통부분렬은 a, m이고 길이는 2이다. 최장공통부분렬을 푸는 알고리듬을 구하시오. 그 알고리듬은 $O(MN)$시간에 실행될수 있어야 한다.

*10-55. **패턴대조문제**(*Pattern matching problem*)는 다음과 같다. 본문의 문자렬 s와

패턴 P가 주어 지면 S안에서 P =1의 첫번째 출현을 구하시오. **근사패턴대조***(Approximate Pattern matching)*는 다음의 세가지 형태의 k개의 오유들을 허용한다;

① 기호는 P에 없지만 S에 있을수 있다.
② 어떤 기호는 P에 있지만 S에 없을수 있다.
③ P와 S는 위치에서 다를수 있다.

례하면 문자렬 《data structures txtborpk》에서 기껏해서 3개의 오유를 허용하면서 패턴《textbook》를 탐색하려고 한다면 정합을 찾아 낼수 있다(e를 삽입하고 r를 o로 바꾸고 P를 삭제하시오.). 근사문자렬대조문제를 푸는 $O(MN)$알고리듬을 구하시오. 여기서 M=|P|이고 N=|S|이다.

***10-56.** 한가지 형태의 **배낭채우기문제**(*knapsack problem*)는 다음과 같다. 옹근수들의 모임
$A=a_1\ a_2\ \cdots a_N$ 과 옹근수 k가 주어 졌다고 한다.
그것의 합이 정확히 k로 되는 A의 부분모임이 존재하겠는가?
ㄱ. 배낭채우기문제를 $O(NK)$시간에 풀어 내는 알고리듬을 작성하시오.
ㄴ. 이것은 왜 $P=NP$를 증명하지 못하는가?

***10-57.** (내리순서로)가격이 $c_1, c_2, \cdots c_n$쎈트인 **동전교환문제**에 대한 통화체계가 주어졌다고 하자.
ㄱ. K쎈트를 교환하는데 필요되는 동전의 최소개수를 계산하는 알고리듬을 구하시오.
ㄴ. K쎈트를 교환하는 여러가지 방법들의 수를 계산하는 알고리듬을 구하시오.

***10-58.** 장기판(8×8)우에서 8녀왕(*eight queens*)들을 배치하는 문제를 고찰하자. 2명의 녀왕들은 같은 행, 같은 렬 즉 같은 대각선상(필수적으로 중요하지는 않다.)에 놓이면 서로를 공격한다고 한다.
ㄱ. 장기판에 8녀왕의 공격을 시도하지 않도록 배치하는 우연화된 알고리듬을 구하시오.
ㄴ. 같은 문제를 풀기 위한 역추적알고리듬을 구하시오.
ㄷ. 두 알고리듬을 실현하고 실행시간을 비교하시오.

***10-59.** 장기경기에서 R행 C렬의 기사는 $1 \leqq R' \leqq B$행과$1 \leqq C' \leqq B$렬에로 움직일수 있는데(여기서 B는 장기판의 크기) 다음 식들이 성립된다.

$|R-R'|=2$ 이고 $|C-C'|=1$ 혹은 $|R-R'|=1$ 이고 $|C-C'|=2$
기사의 순회경로는 시작점에로 되돌아 오기전까지 모든 4각형들을 정확히

한번 방문하는 이동들의 서렬이다.

ㄱ. B가 기수이면 기사의 순회경로는 존재하지 않는다는것을 증명하시오.

ㄴ. **기사의 순회경로**(*knight's tour*)를 찾는 역추적알고리듬을 작성하시오.

10-60. 프로그람 10-19에 보여 준 비순환그라프에서 s부터 t사이의 **무게붙은최단경로**를 탐색하는 재귀알고리듬을 고찰하자.

ㄱ. 이 알고리듬이 왜 일반그라프에 대해서는 처리하지 못하는가?

ㄴ. 비순환그라프들에 대하여 이 알고리듬이 완료된다는것을 증명하시오.

ㄷ. 이 알고리듬의 최악의 경우의 실행시간은 얼마인가?

```
Distance Graph::shortest( s, t )
{
Distance dt;, tmp;
if(   s ==     t   )
return 0;

dt; =  ∞
                for each vertex v adjacent to s
   {
   tmp = shortest( v, t );
       if(cs, v + tmp < dt; )
             dt =    cs, v + tmp;
   }
   return    dt;
}
```

프로그람 10-19. 재귀적인 최단경로알고리듬

10-61. A를 0과 1의 N×N행렬이라고 하자. A의 부분행렬 S는 정방형행렬을 형성하는 어떤 린접입력점들의 그룹이다.

ㄱ. A에서 1의 행렬들중 최대부분행렬의 크기를 구하는 $O(N^2)$알고리듬을 설계하시오. 례하면 아래의 행렬에서 최대부분행렬은 4×4정방행렬이다.

10111000
00010100
00111000
00111010
00111111
01011110
01011110
00011110

**ㄴ. S가 정방형이 아니라 장방형일수도 있다고 할 때 부분문제(ㄱ)를 다

시 설계하시오. 최대크기는 면적으로 측정된다.

10-62. 콤퓨터가 곧 이기게 되는 이동점을 찾아도 그것은 승리를 담보하는 또 다른 이동점을 다시 찾게 된다면 전자의 이동점은 취하지 않는다. 앞에서 본 일부 장기프로그람들은 승리가 검측될 때에 위치반복에 빠져 들어 상대가 비김을 선포하게 만들수 있다는 문제점을 안고 있었다. 세목놓기유희에서는 이것이 문제로 되지 않는바 왜냐하면 이 프로그람은 종국적으로 승리할 것이기때문이다. 세목놓기알고리듬을 수정하여 승리위치가 발견될 때 항상 승리에로 인도하는 가장 작은 이동을 취하도록 하여라. 이것은 COMP_WIN에 9준위의 깊이를 추가하여 가장 빠른 승리가 가장 높은 값을 주도록 하면 될것이다.

10-63. 5×5세목놓기유희에서 하나의 렬에 4개의 목이 있으면 승리하는 프로그람을 작성하시오. 최종매듭들을 탐색해 낼수 있는가?

10-64. 보글의 유희는 문자들의 살창과 어떤 단어목록으로 이루어 진다. 문제는 2개의 린접문자들은 살창에서 린접하고 있어야 하며(즉 north,south,east,west 매개가) 살창안의 매개 항목은 기껏해서 단어당 한번 리용될수 있다는 제한을 만족시키면서 살창안에서 단어들을 탐색해 내는것이다. 보글유희를 노는 프로그람을 작성하시오.

10-65. MAXIT유희를 노는 프로그람을 작성하시오. 유희판은 경기시작시에 임의로 배치된 수들의 N*N살창으로 표현된다. 한 위치를 초기의 현재 위치로 표적한다. 두 선수는 차례를 번갈아 한다. 매개 차례돌림에서 한 선수는 현재 렬 혹은 행에서 살창요소를 선택하여야 한다. 선택된 위치의 값은 그 선수의 점수에 더해 지며 그 위치가 현재 위치로 되고 다시 선택될수 없다. 선수들은 현재행과 렬안의 모든 살창요소들이 선택될 때까지 번갈아 가게 되며 그 시점에서 경기가 끝나는 가장 높은 점수의 선수가 승리한다.

10-66. 6×6판에서 진행하였던 오쩰로는 검은쪽에 대하여 승리를 이룩하였다. 프로그람을 작성해서 이것을 증명하시오. 량쪽에서의 유희가 최적이라면 마지막점수는 얼마인가.

참고문헌

하프만부호에 관한 초기론문은 [22]이다. 이 알고리듬의 여러가지 변화된 내용들은 [30],[33],[34]에서 설명하였다. 또 하나의 일반적인 압축계통은 Ziv-Lempl부호화이다.[60],[61]. 여기서는 부호가 고정길이를 가지지만 기호가 아니라 문자렬들을 표현한

다. [8]과 [36]은 보편적인 압축계통에 대한 훌륭한 연구론문이다.

상자채우기경험규칙의 분석은 Johnson의 박사론문에서 처음 진행되였는데 [23]으로 출판되였다. 련습문제 10.8에 준 직결상자채우기를 위한 개선된 더 낮은 한계는 [57]에서 취한것인바 이 결과는 [37]과 [55]에서보다 훨씬 개선된것이다. [49]는 직결상자채우기에 대한 또 다른 처리방법을 서술한다.

정리 10-7은 [7]에서 취한것이다. 최단점들에 대한 알고리듬은 [50]에서 찾아 볼수 있다. [52]는 통행료금소재구축문제와 그것의 응용들을 서술한다. 지수적인 최악의 경우의 입력은 [59]에서 주어 졌다. 콤퓨터기하학이라는 상대적으로 새 분야에 관한 2개의 이전의 책들은 [14]와 [45]이다. [41]과 [42]는 더 최근의 결과들을 담고 있다. [2]에는 MIT에서 시행된 계산기하학에 관한 강의내용이 포함되여 있는바 거기에는 극히 많은 문헌결과들이 들어 있다.

선형시간선택알고리듬은 [9]에서 찾아 볼수 있다. [17]은 1.5N의 기대된 비교회수로 중간값을 탐색하는 표본화처리방법을 설명한다. $O(N^{1.59})$곱하기는 [24]에서 취한것이다. 이것의 일반화는 [16]과 [26]에서 설명하였다. 스트라센의 알고리듬은 짧은 론문[53]에서 찾아 보시오. 이 론문에서는 결과만 주고 있다. Pan[43]은 몇개의 나누기와 정복알고리듬을 주고 있는데 이것들중 하나가 련습문제 10-27이다. 잘 알려 진 한계는 $O(N^{2.376})$인데 이것은 Coppersmith와 Winograd[13]에 서술되여 있다.

동적계획법에 관한 고전적문헌들은 [5]와 [6]이다. 행렬순서화문제는 [9]에서 처음 연구되였다. 이 문제를 $O(N\log N)$시간에 풀수 있다는것이 [21]에서 설명되였다.

최적2진탐색나무의 구축에 관한 $O(N^2)$알고리듬은 Kunth[27]에서 제기되였다. 모든 쌍최단경로알고리듬은 Floyd[6]에서 취한것이다. 리론적으로 더 좋은 $O(N^3(\log\log N/\log N)^{1/3})$알고리듬은 Fredman[18]에서 주어 진것인데 분명히 실천적인것이 아니다. 약간 개선된 한계(1/3대신에 1/2를 가지는)는 [54]에서 주어 졌는데 관련된 결과들에 대해서는 [3]도 볼 필요가 있다. 어떤 조건하에서는 동적계획법의 실행시간이 자동적으로 N 혹은 그이상의 인자로 개선될수 있다. 이것을 련습문제 10-33, [15], [58]에서 설명하였다.

란수발생기에 대한 설명은 참고서[44]에서 취급한다. 파크와 밀러는 Schrage[51]가 쓴 책에서도 실행을 할수 있도록 내용을 구성하였다. 건너뛰기목록은 Pugh의 책[46]에 의해 서술된다. 다시말하면 treap라고 말할수 있는데 제12장에서 서술되였다. 란수화에서 1차검사알고리듬은 Miller가 쓴 책[38]과 Rabin이 쓴 책[48]에 서술된다. 이 정리는 알고리듬이 헛 동작하게 되는 A의 수는 기껏해서 (N-9)/4라는것이며 이것은 Monier가 쓴 책[39]에서 설명되였다. 다른 란수화알고리듬은 책[47]에서 설명된다. 란수기법에 대하여 더 많은 실례를 준 책은 [21],[25] [40]이다.

α-β가지자르기에 대하여 더 많은 정보를 준 책은 [1],[28],[31]이다. 서양장기와 살창무늬서양장기방법들을 서술한 프로그람은 Othello에 의해 작성되였고 Othello는 세계급의 지위를 차지하게 되였다. Othello프로그람은 책[35]에 준다. 이 책에서는 콤퓨터유희에 대한 내용도 출판되였는데 이 출판은 사색의 보물고이다. 이 책에서는 또한 동적계획법을 리용하는 방법을 주었다. 이 방법은 몇개의 쪽이 장기판에 남았을 때 서양장기의 마지막판을 완전히 풀수 있게 한다.

련습문제 10-41은 [8]에서 풀리여 진다. 련습문제 10-47의 풀이는 문헌[12]에서 준다. 이 알고리듬은 최대한 3/2의 최적으로 실행된다. 련습문제 10-52는 문헌[29]에서 설명한다. 련습문제 10-55는 문헌 [56]에 풀려 져 있다. $O(KN)$알고리듬은 문헌[32]에서 준다. 련습문제 10-57은 문헌[11]에서 설명하였다.

1. B. Abramson, "Control Strategies for Two-Player Games,"' *ACM Computing Surveys,* 21 (1989), 137-161.
2. A. Aggarwal and J. Wein, *Computational Geometry: Lecture Notes for 18.409,* MIT I.aboratory for Computer Science, 1988.
3. N. Alon, Z. Galil, and 0. Margalit, "On the Exponent of the All-Pairs-Shortest Path Problem," *Proceedings of the Thirty-Second Annual Symposium on the foundations of Computer Science,* (1991), 569-575.
4. T. Bell, I. H. Witten, and J. G. Cleary, "Modeling for Text Compression," ACM *Computing Surveys,* 21 (1989), 557-591.
5. R. E. Bellman, *Dynamic Programming,* Princeton University Press, Princeton, N.J., 1957.
6. R. E. Bellman and S. E. Dreyfus, *Applied Dynamic Programming,* Princeton University Press, Princeton, N.J., 1962.
7. J. L. Bentley, D. Haken, anclJ. B. Saxe, "A General Method for Solving divide-and-Conquer Recurrences," *SIGACT News, 12* (1980), 36-44.
8. G. S. Bloom, "A Counterexample to the Theorem of Piccard," *Journal of Combinatorial Theory A (*1977), 378-379.
9. M. Blum, R. W. Floyd, V. R. Pratt, R. I.. Rivest, and R. E. Tarjan, "Time Bounds for Selection," *Journal of Computer and System Sciences, 7* (1973), 448-461.
10. A. Borodin and J. I. Munro, *The Computational Complexity of Algebraic and Numerical Problems,* American Elsevier, New York, 1975.
11. L. Chang and J. Korsh, "Canonical Coin Changing and Greedy Solutions," *Journal of the ACM,* 23 (1976), 418-422.

12. N. Christofides, "Worst-case Analysis of a New Heuristic for the Traveling Salesman Problem," *Management Science Research Report #388,* Carnegie-Mellon University, Pittsburgh, PA, 1976.
13. D. Coppersmith and S. Winograd, "Matrix Multiplication via Arithmetic Progressions," *Proceedings of the Nineteenth Annual ACM Symposium on the Theory of Computing* (1987), 1-6.
14. H. Edelsbrunner, *Algorithms in Combinatorial Geometry,* Springer-Verlag, Berlin, 1987.
15. D. Eppstein, Z. Galil, and R. Giancarlo, "Speeding up Dynamic Programming," *Proceedings of the Twenty-ninth Annual IEEE Symposium on the Foundations of Computer Science,* (1988), 488-495.
16. R. W. Floyd, "Algorithm 97: Shortest Path," *Communications of the ACM,* 5 (1962), 345.
17. R. W. Floyd and R. L. Rivest, "Expected Time Bounds for Selection," *Communications of the ACM,* 18 (1975), 165-172.
18. M. L. Fredman, "New Bounds on the Complexity of the Shortest Path Problem," *SIAM journal on Computing,* 5 (1976), 83-89.
19. S. Godbole, "On Efficient Computation of Matrix Chain Products," *IEEE Transactions on Computers,* 9 (1973), 864-866.
20. R. Gupta, S. A. Smolka, and S. Bhaskar, "On Randomization in Sequential and Distributed Algorithms," ACM *Computing Surveys,* 26 (1994), 7-86.
21. T. C. Hu and M. R. Shing, "Computations of Matrix Chain Products, Part I," *SIAM journal on Computing.* 11 (1982), 362-373.
22. D. A. Huffman, "A Method for the Construction of Minimum Redundancy Codes," *Proceedings of the IRE,* 40 (1952), 1098-1101.
23. D. S. Johnson, A. Demcrs, J. D. Ullman, M. R. Garey, and R. I-. Graham, "Worst-case Performance Bounds for Simple One-Dimensional Packing Algorithms," *SIAM journal on Computing,* 3 (1974), 299-325.
24. A. Karatsuba and Y. Ofman, "Multiplication of Multi-digit Numbers on Automata," *Doklady Akademii Nauk SSSR,* 145 (1962), 293-294.
25. D. R. Karger, "Random Sampling in Graph Optimization Problems," Ph.D. thesis, Stanford University, 1995.
26. D. E. Knuth, *The Art of Computer Programming, Vol 2: Seminumerical Algorithms,* 3d ed., Addison-Wesley, Reading, Mass., 1998.
27. D. E. Knuth, "Optimum Binary Search Trees," *Acta Informatica,* 1 (1971), 14-25.
28. D. E. Knuth, "An Analysis of Alpha-Beta Cutoffs," *Artificial Intelligence,* 6 (1975),

293-326.

29. D. E. Knuth, *TfX and Metafont, New Directions in Typesetting,* Digital Press, Bedford, Mass., 1981.
30. D. E. Knuth, "Dynamic Huffman Coding," *Journal of Algorithms,* 6 (1985), 163-180.
31. D. E. Knuth and R. W. Moore, "Estimating the Efficiency of Backtrack Programs," *Mathematics of Computation,* 29 (1975), 121-136.
32. G. M. Eandau and U. Vishkin, "Introducing Efficient Parallelism into Approximate String Matching and a New Serial Algorithm," *Proceedings of the Eighteenth Annual* ACM *Symposium on Theory of Computing* (1986), 220-230.
33. E. E. Earmore, "Height-Restricted Optimal Binary Trees," *SIAM journal on Computing,* 16(1987), 1115-1123.
34. L. E. Earmore and D. S. Hirschberg, "An East Algorithm for Optimal Eength-Eimited Huffman Codes," *Journal of the ACM,* 37 (1990), 464-473.
35. K. Fee and S. Mahajan, "The Development of a World Class Othello Program," *Artificial Intelligence,* 43 (1990), 21-36.
36. 0. A. Leiewcr and D. S. Hirschherg, "Data Compression," *ACM Computing Surveys,* 19 (1987), 261-296.
37. F. M. Liang, "A Lower Bound for On-line Bin Packing," *Information Processing Letters,* 10 (1980), 76-79.
38. G. L. Miller, "Riemann's Hypothesis and Tests for Primality," *journal of Computer and System Sciences,* 13 (1976), 300-317.
39. I.. Monier, "Evaluation and Comparison of Two Efficient Probabilistic Primality Testing Algorithms," *Theoretical Computer Science,* 12 (1980), 97-108.
40. R. Motwani and P. Raghavan, *Randomized Algorithms,* Cambridge University Press, New York(1995).
41. K. Mulmuley, *Computational Geometry: An Introduction Through Randomized Algorithms,* Prentice-Hall, Englewood Cliffs, N.J. (1994).
42. J. O'Rourke, *Computational Geometry in* C, Cambridge University Press, New York (1994).
43. V. Pan, "Strassen's Algorithm is Not Optimal," *Proceedings of the Nineteenth Annual IEEE Symposium on the Foundations of Computer Science* (1978), 166-176.
44. S. K. Park and K. W. Miller, "Random Number Generators: Good Ones arc Hard To Find," *Communications of the ACM.* 31 (1988), 1192-1201. (See also *Technical Correspondence,* in 36 (1993) 105-110).
45. F. P. Preparata and M. I. Shames, *Computational Geometry: An Introduction,* Springer-Verlag, New York, 1985.

46. W. Pugh, "Skip Lists: A Probabilistic Alternative to Balanced Trees," *Communications of the ACM,* 33 (1990), 668-676.
47. M. 0. Rabin, "Probabilistic Algorithms," in *Algorithms and Complexity, Recent Results and New Directions* (J. F. Traub, cd.). Academic Press, New York, 1976, 21-39.
48. M. 0. Rabin, "Probabilistic Algorithms for Testing Primality," *Journal of Number Theory,* 12(1980), 128-138.
49. P. Ramanan, D. J. Brown, C. C. I.ee, and D. T. Lee, "On-line Bin Packing in I.inear Time," *Journal of Algorithms,* 10 (1989), 305-326.
50. M. I. Shamos and D. Hoey, "Closest-Point Problems," *Proceedings of the Sixteenth Annual IEEE Symposium on the Foundations of Computer Science* (1975), 151-162.
51. L. Schrage, "A More Portable FORTRAN Random Number Generator," ACM *Transactions on Mathematics Software, 5* (1979), 132-138.
52. S. S. Skiena, W. D. Smith, and P. Lemke, "'Reconstructing Sets From Interpoint Distances," *Proceedings of the Sixth Annual ACM Symposium on Computational Geometry* (1990), 332-339.
53. V. Strassen, "Gaussian Elimination is Not Optimal," *Mumerischc Mathematik,* 13 (1969), 354-356.
54. T. Takaoka, "A New Upper Bound on the Complexity of the All-Pairs Shortest Path Problem," *Information Processing letters,* 43 (1992), 195-199.
55. A. van VIeit, "An Improved Lower Bound for On-Line Bin Packing Algorithms," *Information Processing Letters,* 43 (1992), 277-284.
56. R. A. Wagner and M. J. Fischer, "The String-to-String Correction Problem," *Journal of the ACM,* 21 (1974), 168-173.
57. A. C.Yao, "New Algorithms for Bin Packing," *journal of the* ACM, 27 (1980), 207-227.
58. F. F. Yao, "Efficient Dynamic Programming Using Quadrangle Inequalities," *Proceedings of the Twelfth Annual ACM Symposium on the Theory of Computing* (1980), 429-435.
59. Z. Zhang, "An Exponential Example for a Partial Digest Mapping Algorithm," *Journal of Computational Molecular Biology,* 1 (1994), 235-239.
60. J. Ziv and A. Lempel, "A Universal Algorithm for Sequential Data Compression," *IEEE Transactions on Information Theory* IT23 (1977), 337-343.
61. J. Ziv and A. Lempel, "Compression of Individual Sequences via Variable-rate Coding," *IEEE Transactions on Information Theory* IT24 (1978), 530-536.

제11장. 상환분석

이 장에서는 제4장과 제6장에서 보여 준 몇가지 개선된 자료구조들에 대한 실행시간을 해석하려고 한다. 특히 임의의 M개 연산에 대한 최악의 경우 실행시간을 고찰하려고 한다. 이것은 최악의 경우의 연산시간한계가 임의의 단일한 연산으로부터 주어 지는 보다 일반적인 해석과는 다르다.

실례로 AVL나무는 연산당 최악의 경우 $O(\log N)$시간으로 표준나무연산을 제공한다는데 대하여 보았다. AVL나무들은 약간 실현하기가 복잡한데 그것은 여러가지 경우가 있을뿐아니라 높이균형정보를 보관하여야 하고 정확히 갱신하여야 하기때문이다. AVL나무가 쓰이는 리유는 불균형탐색나무에서는 $\Theta(N)$연산의 렬이 ㅠㅠ$\Theta(N^2)$시간을 요구하는데 이것이 비경제적이라는데 있다. 탐색나무에서는 연산의 $O(N)$실행시간이 문제인것이 아니다. 기본문제는 이 연산이 반복출현한다는것이다. 펼친나무들은 흥미 있는 다른 방안을 준다. 비록 임의의 연산이 여전히 $\Theta(N)$시간을 요구하지만 불필요한 연산은 반복출현하지 않는다. 임의의 M개의 연산렬에 대하여 최악의 경우 $O(M\log N)$의 실행시간을 요구한다는것을 증명할수 있다. 그러므로 이 자료구조는 매 연산당 평균 $O(\log N)$시간으로 동작한다. 이것을 **상환된 시간한계**(*amortized time bound*)라고 한다.

상환된 시간한계는 대응하는 최악의 경우의 시간한계보다는 더 작은데 이것은 임의의 간단한 조작에 대한 담보가 없기때문이다. 만일 조작렬에 대하여 한계가 같고 동시에 자료구조가 단순하다면 단순연산에 대한 한계는 얼마 크지 않을것이다. 실례로 2진탐색나무는 매 연산당 $O(\log N)$의 평균시간이 걸리지만 여전히 M개의 조작렬은 $O(MN)$시간이 걸리게 할수 있다.

상환된 한계를 이끌어 낸다는것은 하나만이 아니라 전체 조작렬에 대하여 주시할것을 요구하므로 해석은 더 복잡하리라고 예상된다. 이것이 일반적으로 사실이라는것을 보게 된다.

이 장에서는 다음의 내용을 고찰한다.

- 2항대기렬연산을 해석한다.
- 경사더미를 해석한다.
- 피보나치더미를 도입하고 해석한다.
- 펼친나무를 해석한다.

제1절. 무관계알아맞추기

다음과 같은 알아맞추기문제를 고찰하자. 두 마리의 새끼고양이가 축구장의 량쪽 끝에 100야드의 거리를 두고 놓여 있다. 그것들은 상대방을 향하여 분당 12야드의 속도로 달린다. 같은 시각에 그것들의 엄지가 마당의 한 끝에 있다. 어미는 분당 100야드의 속도로 달릴수 있다. 그 엄지는 새끼들이 마당가운데서 만나게 될 때까지 속도손실이 없이 한 새끼로부터 다른 새끼에로 달린다. 엄지는 얼마만큼 달렸겠는가?

이 알아맞추기를 수동적인 방법으로 계산해 내는것은 그리 어렵지 않다. 보다 구체적인 계산은 독자들에게 맡기지만 기대되는것은 이 계산이 **무한등비수렬의 합계산문제**라는것이다. 간단한 계산으로 어떤 답을 얻을수 있으나 훨씬 더 간단한 답은 보충변수 즉 시간변수를 도입하여 얻을수 있다는것을 알수 있다. 새끼들은 100야드 떨어 져 있고 분당 20야드의 합성속도로 서로 접근하므로 마당한가운데점에 도달하는데 5분 걸린다.

엄지가 분당 100야드 달리므로 그의 총 거리는 500야드이다. 이 알아맞추기는 대체로 어떤 문제를 직접적인 방법보다 간접적인 방법으로 푸는것이 더 쉽다는것을 보여 주고 있다. 우리가 수행할 상환분석은 역시 이러한 내용을 담고 있다. 포텐샬변수를 도입하여 그렇게 하지 않으면 아주 얻기 힘들것 같은 결과들도 쉽게 얻어 내는 방법들을 보게 될것이다.

제2절. 2항대기렬

우리가 고찰하려는 첫번째 자료구조는 제6장의 **2항대기렬**(*binomial queues*)인데 이제 이것을 보기로 하자. **2항나무**(*binomial tree*) B_0은 한개 매듭나무이고 k>0에 대하여 2항나무 B_k는 2개의 2항나무 B_{k-1}과 결합하여 얻어 진다는것을 상기하자. 2항나무 B_0부터 B_4까지를 그림 11-1에 보여 주었다.

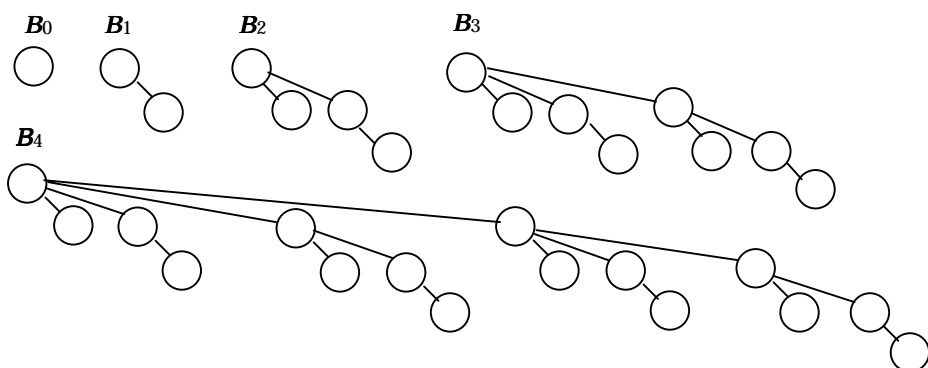
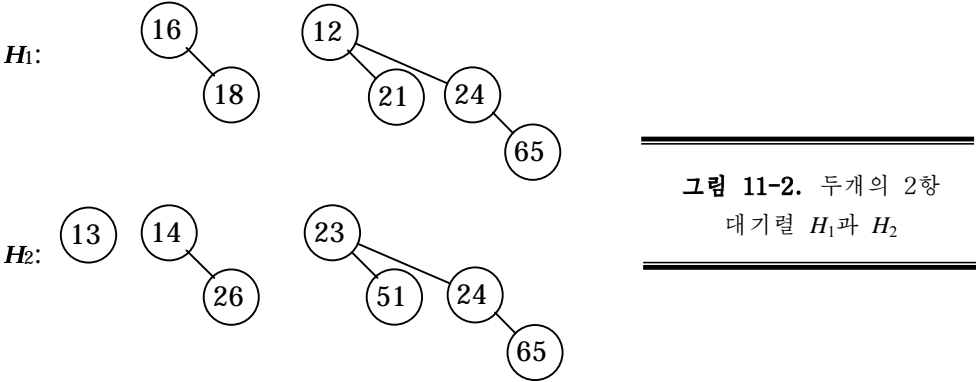



그림 11-1. 2 항나무 B_0, B_1, B_2, B_3, B_4

2항나무안에서 매듭의 **위수**(*rank*)는 새끼매듭의 개수와 같은데 특히 B_k의 뿌리매듭의 위수는 k이다. 2항대기렬은 더미가 순서화된(heap-ordered) 2항나무들의 집합인데 여기서 임의의 k에 대하여 기껏해서 한개의 2항나무 B_k가 있을수 있다. 2개의 2항나무대기렬 H_1과 H_2를 그림 11-2에 보여 주었다.

그림 11-2. 두개의 2항 대기렬 H_1과 H_2

가장 중요한 연산은 **병합연산**(*merge*)이다. 2개의 2항대기렬을 병합하기 위하여 2진옹근수들의 더하기와 류사한 연산이 수행된다. 임의의 단계에서 두개의 우선권대기렬이 B_k나무를 포함하는가 안하는가 그리고 B_k나무가 이전의 단계에서 왔는가 안왔는가에 관계되는 령,하나, 둘 또는 세개의 B_k나무를 얻을수 있다. 령 혹은 한개의 B_k나무가 있다면 이것은 결과적인 2항대기렬에서 나무처럼 배치된다. 두개의 B_k나무가 있다면 그것들은 B_{k+1}나무로 결합되여 다음 단계로 넘어 간다. 세개의 B_k나무가 있다면 한개는 그 2항대기렬안의 나무로 배치되고 다른 2개가 결합되여 넘겨 진다. H_1과 H_2를 련결한 결과를 그림 11-3에 보여 주었다.

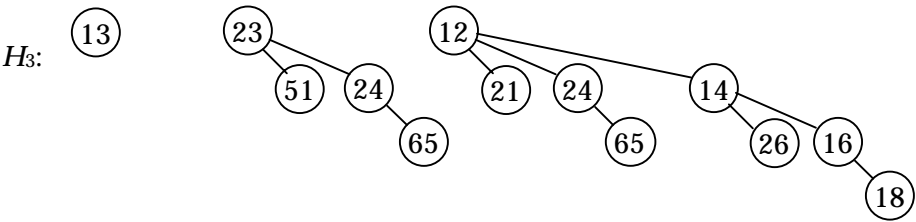



그림 11-3. 2항대기렬 H_3: H_1와 H_2의 련결

삽입연산은 한개 매듭의 2항대기렬을 만들고 한번의 병합을 수행하여 실행된다. 이 조작을 수행하는데 걸리는 연산시간은 $M+1$인데 여기서 M은 그 2항대기렬에 들어 있지 않는 가장 작은 형태의 2항나무 B_M을 표현한다. 따라서 B_0나무에는 있지만 B_1나무에는 없는 2항대기렬에로의 삽입연산은 두개의 단계를 요구한다. 최소나무의 삭제는 최소나무를 제거하고 원래의 2항대기렬은 두개의 2항대기렬로 분해하는 방법으로 수행하며 그다

음 이 두개의 2항나무는 병합된다. 이 연산에 대한 보다 간단한 설명은 제6장에서 취급하였다.

가장 간단한 문제부터 고찰하자. N개의 요소들로 이루어 진 2항대기렬을 구축한다고 하자. N개 요소들의 2진더미를 구축하는데 시간한계 $O(N)$로 수행된다는데 대해서는 이미 알고 있으므로 2항대기렬에 대하여도 이와 류사한 한계를 예상하고 있다.

명제:

N개의 요소들로 된 2항대기렬에 대한 N번의 삽입연산은 $O(N)$시간안에 성공적으로 진행할수 있다.

우의 명제가 사실이라면 아주 간단한 알고리듬을 준다. 한번의 삽입연산에 대한 경계시간은 $O(\log N)$이므로 이 명제가 사실인지 현재는 명백치 않다. 이 알고리듬이 2진더미에 적용되면 실행시간은 $O(\mathrm{N}\log N)$으로 된다.

이 명제를 증명하기 위하여 직접 계산해 볼수 있다. 실행시간을 측정하기 위하여 매 삽입연산시간은 하나의 시간단위에 매 련결단계의 여분의 시간단위를 더한것으로 정의한다. 모든 삽입연산들에 대하여 이 값들을 더하면 총체적인 실행시간이 얻어 진다. 이 총시간은 N단위에 련결단계들의 총수를 더한것이다. 첫번째, 세번째, 다섯번째 그리고 홀수번호의 단계들은 모두 련결단계를 요구하지 않는데 이것은 삽입연산에서 B_0이 존재하지 않기때문이다. 그러므로 절반의 삽입연산들은 련결단계를 요구하지 않는다. 삽입연산들의 1/4만이 하나의 련결단계를 요구한다(두번째, 여섯번째, 열번째 등). 여덟번째는 2개의 련결단계를 요구한다. 그리고 그 이후는 이런 식으로 반복될것이다. 이 모든것을 합하면 련결단계의 수를 N으로 제한할수 있으며 따라서 명제가 증명된다. 이러한 맹목적인 계산방식은 삽입연산이 많은 연산렬을 분석하려 할 때에는 도움을 주지 못하므로 이 결과를 증명하기 위하여 다른 근사방법을 리용하려고 한다.

삽입연산결과를 고찰하자. 삽입연산시 B_0나무가 없으면 우와 같은 계산을 리용하여 삽입연산의 시간은 총 한단위의 시간으로 된다. 결과 B_0나무가 있게 되며 따라서 2항나무들의 수림에 한개의 나무가 추가되였다. B_0나무는 있지만 B_1나무가 없다면 삽입연산은 2단위로 된다. 새로운 수림은 B_1나무를 가지지만 B_0나무는 가지지 않으므로 이 수림에서 나무개수는 변하지 않는다. 3단위로 되는 삽입연산은 B_2나무를 만들지만 B_0나무와 B_1나무를 파괴하며 결과 숲에서 한개 나무를 손실당한다. 사실 c단위로 되는 삽입연산이 수림에서 $2-c$개 나무들의 증대그물을 형성하게 된다. 그것은 B_{c-1}개 나무가 만들어 져도 $B_i(0 \leqq i < c-1)$나무들은 제거되기때문이다. 그러므로 시간이 많이 드는 삽입연산들은 나무들을 제거하고 반대로 시간이 덜 드는 삽입연산은 나무들을 만든다.

C_i는 i번째 삽입연산의 연산시간이라고 하자. T_i는 i번째 삽입연산후의 나무들의 개수라고 하자. $T_0=0$은 초기나무들의 개수이다. 그러면 다음의 항등식이 성립한다.

$$C_i+(T_i-T_{i-1})=2 \quad (11\text{-}1)$$

이로부터 다음 식들을 얻는다.

$$C_1+(T_1-T_0)=2$$
$$C_2+(T_2-T_1)=2$$
$$\cdots$$
$$C_{N-1}+(T_{N-1}-T_{N-2})=2$$
$$C_N+(T_N-T_{N-1})=2$$

이 식들을 모두 더하면 대부분의 T_i항들은 소거되고 다음 식이 얻어 진다.

$$\sum_{i=1}^{N} C_i + T_N - T_o = 2N$$

혹은 등가적으로 다음 식을 얻는다.

$$\sum_{i=1}^{N} C_i = 2N - (T_N - T_0)$$

T_0=0이고 N회의 삽입연산후의 나무개수 T_N이 확고하게 부의 값이 아니므로 $T_N - T_0$은 부수가 아니다. 그러므로

$$\sum_{i=1}^{N} C_i \le 2N$$

이다. 따라서 명제가 증명된다.

2항대기렬의 구축(*Build Binomial Queue*)루틴과정에 매번의 삽입연산은 $O(\log N)$의 최악의 경우 시간이 걸리지만 전체 루틴은 기껏해서 $2N$단위시간을 리용하기때문에 이 삽입연산들은 매회의 연산이 2단위시간이상은 걸리지 않는다.

이 실례는 우리가 리용할 일반적기술을 설명한다. 임의의 시각에 자료구조의 상태는 **포텐샬**(*potential*)이라는 함수로 주어 진다. 포텐샬함수는 프로그람에 의하여 보관되는것이 아니지만 해석에 도움을 주는 계산방식이다. 연산들이 할당한 시간보다 더 적게 걸려서 실행될 때 리용되지 않는 시간은 더 높은 포텐샬값으로 보관된다. 실례에서 자료구조의 포텐샬은 단순히 나무의 수이다. 우의 해석에서 할당된 2 단위대신에 하나의 단위만을 리용하는 삽입연산이 있을 때 여분의 단위는 후에 포텐샬에서의 증가로 보관된다. 할당된 시간을 초과하는 연산들이 발생할 때에는 초과시간이 포텐샬에서의 감소로 계산된다. 이것은 포텐샬이 보관값(Savings account)을 표현한다고 볼수 있을것이다. 연산이 그것에 할당된 시간보다 더 작게 진행된다면 그 차는 후에 시간을 더 소비하는 연산에 리용하기 위해 보관된다. 그림 11-4는 삽입연산들의 렬에서 build Binomial Queue에 의해 리용된 루적실행시간을 보여 주었다. 실행시간이 결코 $2N$을 초과하지 않으며 임의의 삽입연산후의 2항대기렬안의 포텐샬이 보관량을 측정한다는데 주의를 돌리시오.

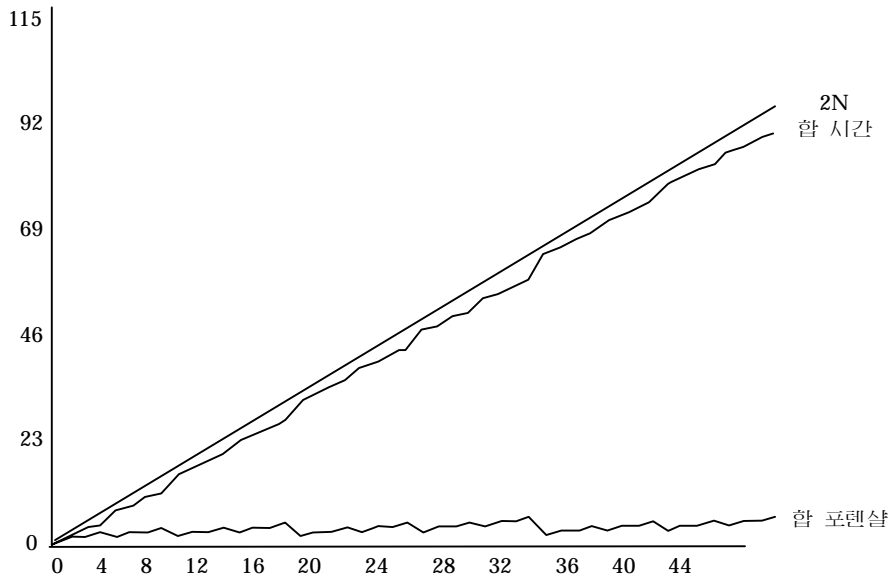



그림 11-4. N개 대기렬의 삽입연산

일단 포텐샬함수가 선택되면 기본식은 다음과 같이 쓸수 있다.

$$T_{actual}+\triangle Potential=T_{amorized} \tag{11-2}$$

연산의 실제시간 T_{actual}은 개개의 연산을 실행하는데 요구되는 정확한 시간량(관측된)을 표현한다. 실례로 2진탐색나무에서 find(x)를 수행하는데 드는 실제시간은 x를 포함하는 매듭의 깊이에 1을 합한것과 같다. 전체 렬에서 기본식을 합하면 그리고 마지막포텐샬이 적어도 초기의 포텐샬만큼 크다면 상환된 시간은 그 렬의 실행과정에 리용된 실제시간에 대한 상한(웃한계)으로 된다. T_{actual}이 연산에 따라 변하지만 $T_{amorized}$는 안정하다는데 대하여 주의해야 한다.

의미 있는 한계를 증명하는 포텐샬함수를 결정하는것은 대단히 의의 있는 과제인바 일반적인 방법은 없다. 우의 설명은 몇가지 규칙을 정하자는것인데 이 규칙들은 좋은 포텐샬함수들이 가지고 있는 속성을 준다.

포텐샬함수는

- 렬의 시작에서 항상 최소값을 가진다고 가정한다. 포텐샬함수를 선택하는 일반적인 방법은 포텐샬함수가 초기에 0을 가지며 항상 부가 아니도록 하는것이다. 우리가 취급하려는 모든 실례들에서는 이 전략을 리용한다.
- 동작시간에 항을 소거하여야 한다. 이 경우에 동작시간이 c였다면 포텐샬의 변동은 $2-c$로 된다. 이것들이 부가될 때 2의 상환시간이 얻어 진다. 이것을 그림 11-5에 보여 주었다. 이제는 2항대기렬연산을 완전히 분석할수 있다.

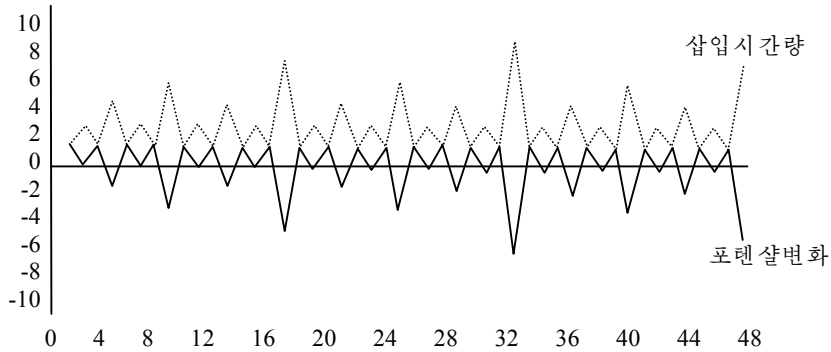



그림 11-5. 대기렬안에서 매개 연산에 대한 삽입연산시간과 포텐샬변화

정리 11-1.
insert, deleteMin, merge의 상환된 실행시간은 2항대기렬에 대하여 각각 $O(1)$, $O(\log N)$, $O(\log N)$이다.

증명:
포텐샬함수(*potential function*)는 나무의 수이다. 초기포텐샬은 0이며 포텐샬은 항상 부수가 아니므로 상환된 시간은 실제시간의 웃한계이다. Insert를 위한 분석은 웃의 증명으로부터 나온다. Merge에 대해서는 N_1과 N_2매듭을 가지는 두개의 나무를 각각 T_1, T_2라고 하자. 이것을 련결하는데 드는 실제시간은 $O(\log(N_1)+\log(N_2))=O(\log N)$이다. 병합후에 기껏해서 $\log N$의 나무들이 있을수 있으며 그러므로 포텐샬은 기껏해서 $O(\log N)$만큼 증가할수 있다. 이것은 상환한계를 $O(\log N)$으로 한다. DeleteMin의 한계도 류사한 방식으로 증명된다.

제3절. 경사더미

2항대기렬의 해석은 극히 단순한 상환해석의 한가지 실례이다. 이제 경사더미를 고찰하자. 많은 실례들에서처럼 정확한 포텐샬함수가 주어 지면 해석은 쉽다. 의미 있는 포텐샬함수를 선택하는것은 힘들다.

경사더미(*skew heap*)에서는 기본연산이 병합이라는것을 상기시킨다. 2개의 경사더미를 병합하기 위해서 그것들의 오른쪽 경로들을 련결하며 이것을 새로운 왼쪽 경로로 만든다. 새 경로우의 매개 매듭(마지막매듭은 제외)에 대하여 이전의 왼쪽 부분나무가 오른쪽 부분나무로 붙어 있다. 새 왼쪽 경로우의 마지막매듭은 오른쪽 부분나무를 가지지 않게 되며 그래서 그것에 한개의 무게를 주는것은 정확치 못하다. 시간한계는 그 나머지

매듭에 관계되지 않으며 이 루틴이 재귀적으로 작성된다면 이것은 아주 자연스러운것이다. 그림 11-6에서는 2개의 경사더미를 병합한 결과를 보여 주었다. 2개의 경사더미 H_1과 H_2가 있는데 그것들 매개의 오른쪽 경로들에 r_1과 r_2의 매듭들이 있다고 하자.

병합하는데 드는 실제시간은 $r_1 + r_2$에 비례하며 이로부터 큰O표기법을 떨구어서 경로의 매 매듭에 대하여 한 단위의 시간을 축적한다. 더미들이 어떤 구조도 가지지 않기때문에 그 더미안의 모든 매듭들이 오른쪽 경로우에 놓이는것은 가능하며 이것은 두 더미를 병합하는데 드는 최악의 경우의 한계를 $\Theta(N)$으로 규정한다(련습문제 11-3은 이런 실례를 구축하는 문제). 두 경사더미를 련결하는데 드는 상환된 시간은 $O(\log N)$이다.

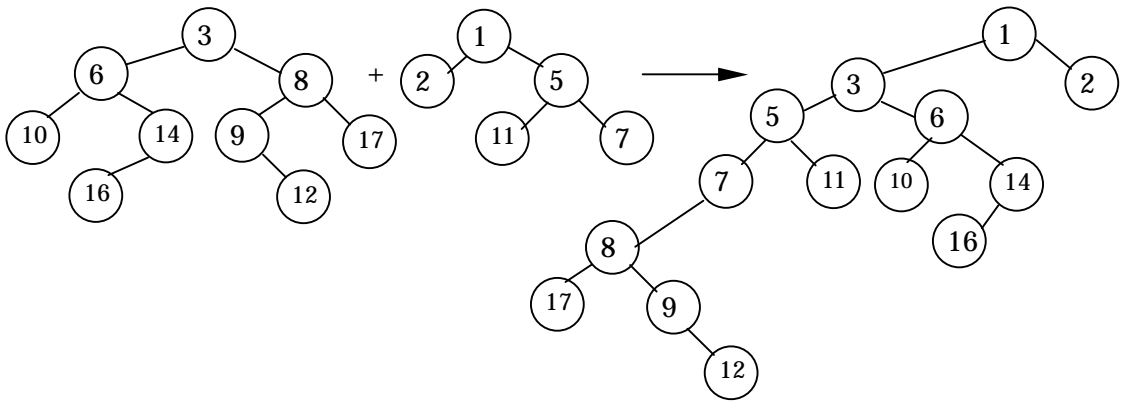



그림 11-6. 두 경사더미들의 병합

경사더미연산들의 효과를 발생하는 어떤 형태의 포텐샬함수가 필요하다. 병합의 효과는 오른쪽 경로우의 매개 매듭이 왼쪽 경로로 움직이게 하는것이며 그것의 이전 왼쪽의 자식매듭은 새로운 오른쪽 자식으로 만든다는것이다. 한가지 방안은 매듭이 오른쪽자식인가 아닌가에 따라 매개 매듭을 오른쪽 매듭 혹은 왼쪽 매듭으로 분류하고 오른쪽매듭들의 수를 포텐샬함수로 리용하는것이다. 문제는 이 포텐샬이 초기에 0이고 항상 부수가 아니라고 하지만 포텐샬이 병합된후에 감소하지 않으며 자료구조에서의 보관을 적당히 반영하지 못한다는것이다. 결과 이 포텐샬함수는 지정된 한계를 증명하는데 리용할수 없다.

비슷한 방안은 어떤 매듭의 오른쪽 부분매듭이 왼쪽 부분매듭보다 더 많은 매듭들을 가지는가 아닌가에 따라 매듭들을 가벼운 매듭 혹은 무거운 매듭으로 분류한다.

정의: 매듭 p의 오른쪽 부분나무의 자손의 개수가 적어도 p의 자손의 수의 절반이라면 매듭 p는 무거운 매듭라고 하며 그렇지 않으면 가벼운 매듭이라고 한다. 매듭의 자손의 개수에는 그 매듭자체도 포함된다는데 주의를 준다

실례로 그림 11-7에서는 경사더미를 보여 주었다. 15, 3, 6, 12, 7의 번호가 붙은 매듭들은 무거운 매듭이며 나머지매듭들은 모두 가벼운 매듭들이다.

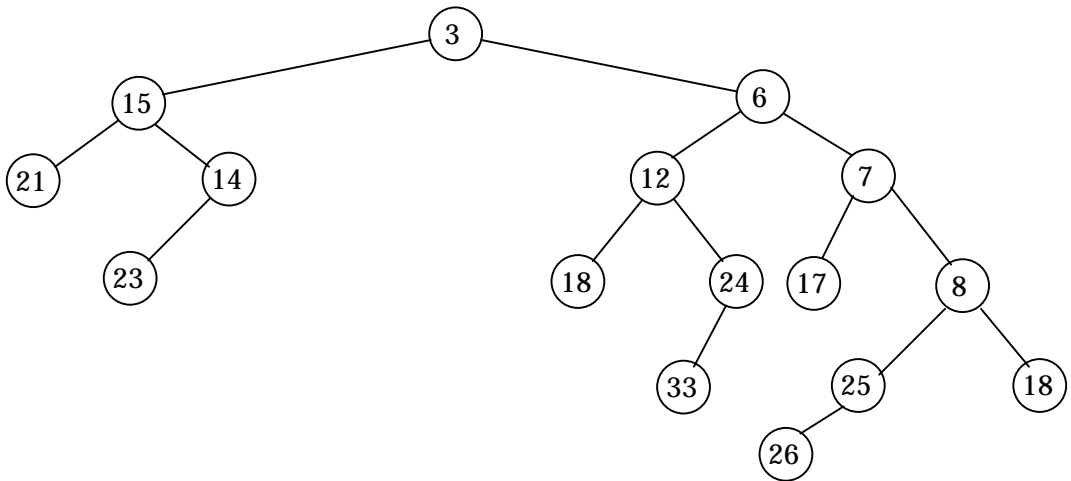



그림 11-7. 경사더미-무거운 매듭은 3, 6, 7, 12, 15

리용하려는 포텐샬함수는 더미(집합)에서 무거운 매듭들의 수이다. 이것은 잘 선택된것처럼 생각되는데 오른쪽 긴경로는 지나친 개수의 무거운 매듭들을 포함한다. 이 경로의 매듭들은 그것들의 자손들을 교체하므로 병합된후에 가벼운 매듭들로 전환된다.

정리 11-2.

두 경사더미들을 병합하는데 드는 상환된 시간은 $O(\log N)$이다.

증명:

H_1과 H_2를 각각 N_1과 N_2의 매듭들을 가지는 두더미라고 하자. H_1의 오른쪽 경로가 l_1개의 가벼운 매듭들과 h_1개의 무거운 매듭들을 가진다고 가정하자. 전체의 매듭수는 $l_1+ h_1$이다. 같은 방법으로 총 매듭이 l_1+h_2일 때 H_2는 l_2개의 가벼운 매듭과 h_2개의 무거운 매듭들을 오른쪽 경로에 가지고 있다.

두 경사더미들을 병합하는데 드는 시간이 그것들의 오른쪽 경로우의 매듭들의 총 개수라는 약속을 따른다면 병합하는데 드는 실제 시간은 $l_1+ l_2+ h_1+ h_2$이다. 이제 무겁고 가벼운 상태가 변할수 있는 매듭들만이 초기에 오른쪽 경로우에 있는 매듭으로 되는데(왼쪽 경로우에 붙어 있다.) 이것은 다른 매듭들을 그것들의 부분나무로 교체하지 않기때문이다. 이것은 그림 11-8의 실례로 보여 주었다.

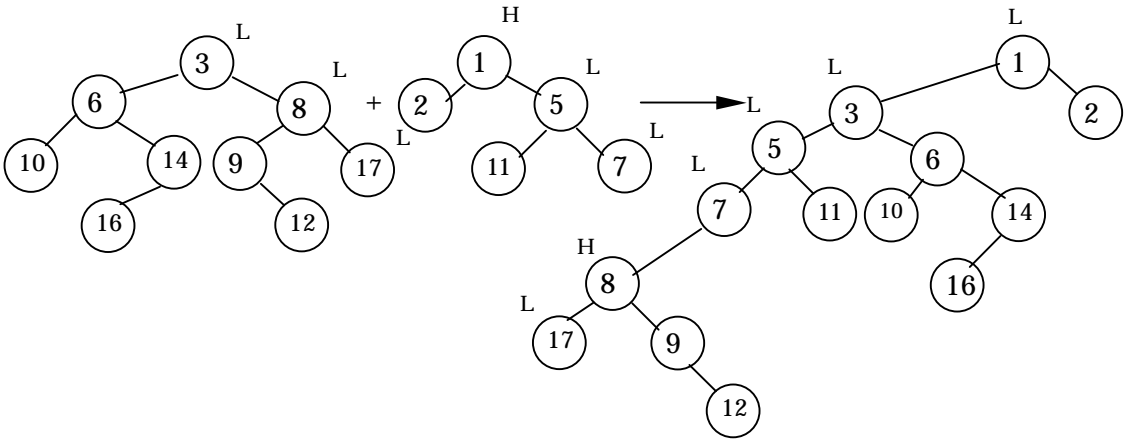



그림 11-8. 련결후에 무겁고 가벼운 상태에서의 변화

무거운 매듭이 초기에 오른쪽 경로우에 있다면 병합후에 그것은 가벼운 매듭으로 되여야 한다. 오른쪽 경로우에 있었던 다른 매듭들은 가벼운 매듭이였는데 무겁게 될수도 있고 안될수도 있는바 웃한계를 증명해야 하기때문에 최악의 경우를 생각해야 할것이며 그래서 그것들은 무겁게 되여 포텐샬을 증가시킨다. 그러면 무거운 매듭들의 개수에서의 무게변화는 기껏해서 $l_1+l_2-h_1-h_2$이다. 실제시간과 포텐샬변화를 합하면 $2(l_1+l_2)$의 상환한계를 얻는다.

이제는 $l_1+l_2 O(\log N)$을 고찰해야 한다. l_1과 l_2도 원래의 오른쪽 경로우의 가벼운 매듭들의 개수이며 가벼운 매듭의 오른쪽 부분나무는 그 가벼운 나무에 뿌리를 둔 나무의 크기보다 절반이하이기때문에 오른쪽 경로우의 가벼운 매듭들의 개수는 기껏해서 $\log N_1+\log N_2$ 즉 $O(\log N)$이라는것을 알수 있다.

이 증명은 초기의 포텐샬이 0이며 포텐샬이 항상 부수가 아니라는것을 고려할 때 완성된다. 이것을 고려하는것이 중요한데 왜냐하면 그렇지 않다면 상환된 시간은 실제시간의 한계로 될수 없고 의미를 잃어 버리기때문이다.

Insert와 deleteMin연산들이 본질적으로는 Merge연산들이므로 그것들도 상환한계가 $O(\log N)$이다.

제4절. 피보나치더미

제9장 제3절 2에서 우선권대기렬을 리용하여 딕스트라의 최단경로알고리듬의 실행시간을 어떻게 $O(|V|^2)$으로 개선할수 있는가를 보았다. DecreasKey연산의 실행시간은 $|E|$, insert와 deleteMin연산의 실행시간을 $|V|$로 된다는것을 관찰하였다. 이 연산들은 기껏해서

$|V|$의 크기를 가진다. 2항더미를 리용하게 되면 이 모든 연산들은 $O(\log|V|)$시간이 걸리며 따라서 딕스트라의 알고리듬에 대한 결과한계는 $O(|E|\log|V|)$으로 작아 질수 있다.

이 시간한계를 더 낮추기 위해서는 decrease key연산을 수행하는데 요구되는 시간이 개선되여야 한다. 제6장 제5절에서 설명하는 **d-더미**(*d-heap*)는 decreaseKey연산은 물론 insert연산에 대하여 $O(\log_d|V|)$의 시간한계를 취하지만 deletemin에 대해서는 $O(d\log_d|V|)$한계를 취한다. $|E|$decreasekey연산의 시간을 $|V|$deletemin연산시간과 평형을 이루도록 d를 선택한다. 그리고 d가 항상 적어도 2이여야 한다는것을 고려하면 d에 대하여 다음과 같이 적당한 선택을 진행될수 있다.

$$d = \max(2, \lfloor |E|/|V| \rfloor)$$

이것은 **딕스트라알고리듬**(*Dijkstra's algorithm*)에 대한 시간한계를 다음의 값으로 개선한다.

$$O(|E|\log_{(2+\lfloor |E|/|V| \rfloor)}|V|)$$

피보나치더미(*Fibonacci heap*)는 $O(\log N)$상환시간을 가지는 deletemin과 remove를 제외하고 모든 기본더미연산들을 $O(1)$상환시간으로 지원하는 자료구조이다. 그러므로 딕스트라의 알고리듬안의 더미연산들이 총 $O(|E|+\log|V|)$시간을 요구한다는것을 알수 있다.

피보나치더미는 다음의 두개의 새로운 개념들을 추가하면 2항대기렬로 일반화된다.

Decreaskey의 다른 실현부: 앞에서 고찰한 방법은 원소를 뿌리를 향하여 우로 올라가면서 추출하는것이다. 이 방법에 대해서는 상환한계를 $O(1)$로 기대하는것이 어려우며 그래서 새로운 방안을 생각한다.

지연병합(*Lazy merging***):** 두개의 더미는 결합되여야 할 요구가 제기되여야 결합된다. 이것도 지연삭제와 류사하다. 지연병합일 때 merge는 시간이 적게 들지만 지연병합이 실제 나무들을 결합하는것이 아니기때문에 deletemin연산도 많은 나무들을 고려하여야 하며 그래서 연산에 시간이 많이 들게 된다. 임의의 한계의 deletemin연산은 선행시간을 취할수 있지만 이전의 merge연산들에는 그 시간을 축적하는것은 항상 가능하다. 특히 시간이 많이 드는 deletemin은 아주 시간이 적게 드는 많은 개수의 merge연산들의 뒤에 놓아야 하며 이렇게 함으로써 여분의 포텐샬을 보관할수 있다.

1. 왼쪽더미들에서 매듭자르기

2진더미에서 decreasekey연산은 매듭에서의 값을 낮추면서 더미차수가 성립될 때까지 이 동작을 뿌리로 올라 가면서 수행한다. 최악의 경우에 이것은 $O(\log N)$시간 걸릴수 있으며 이 시간은 균형나무에서 뿌리로 향하는 가장 긴 경로의 길이에 관계된다.

우선권대기렬을 포함하는 나무가 $O(\log N)$깊이를 가지지 않는다면 이 방법을 쓸수 없다. 례를 들어 이 방법을 왼쪽더미에 적용하면 decreasekey연산은 $\Theta(N)$시간 걸린다. 그 실례를 그림 11-9에 보여 주었다.

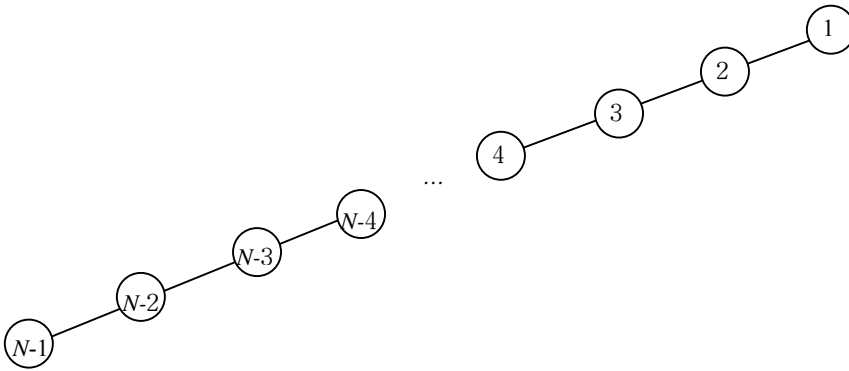



그림 11-9. 추출과정에 N-1을 0으로 감소시키는데 $\Theta(N)$시간이 걸린다.

왼쪽더미인 경우 decreasekey연산에 대한 다른 방법이 필요하다는것을 고찰한다. 그림 11-10에 왼쪽더미를 실례로 주었다. 값 9를 가진 열쇠를 2까지 감소시키려고 한다고 하자. 변화되면 더미차수의 위반이 생긴다는것을 알수 있는데 더미차수의 위반을 그림 11-11에서 파선으로 지적하였다.

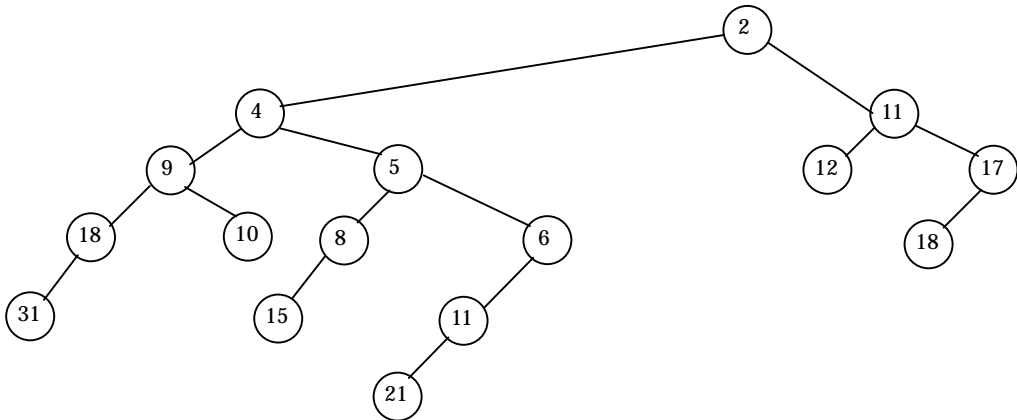



그림 11-10. 간단한 왼쪽더미 H

이 0을 뿌리로 찾아 올라 가는것을 바라지는 않는데 그것은 이미 본것처럼 시간이 많이 드는 경우가 있기때문이다. 대책은 파선에서 더미를 잘라 내여 두개의 나무를 만든 다음 그 두 나무를 후에 하나의 나무로 병합시키는것이다. X를 decreasekey연산이 적용되

고 있는 매듭, P를 그것의 부모마디라고 하자. 잘라 진 이후 2개 나무 즉 뿌리가 X인 H_1과 H_1이 제거된 원래의 나무 T_2가 얻어 진다. 이 상태를 그림 11-12에 보여 주었다.

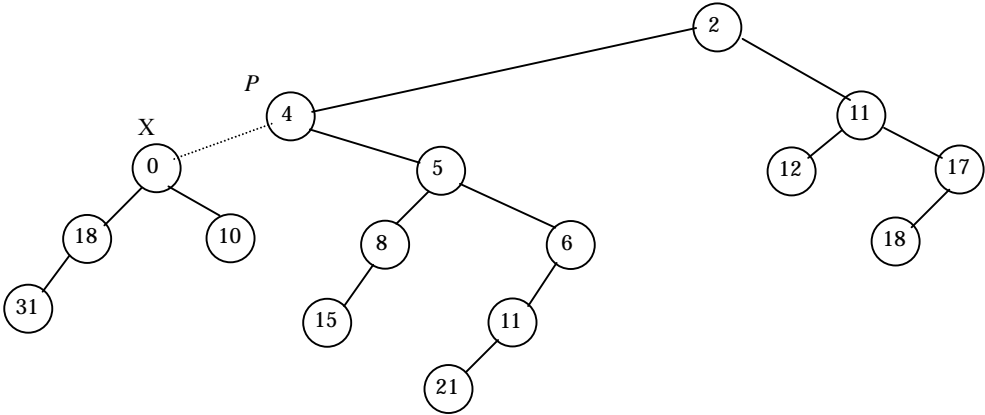



그림 11-11. 9를 0으로 감소시키는것은 더미차수의 낮추기를 발생시킨다.

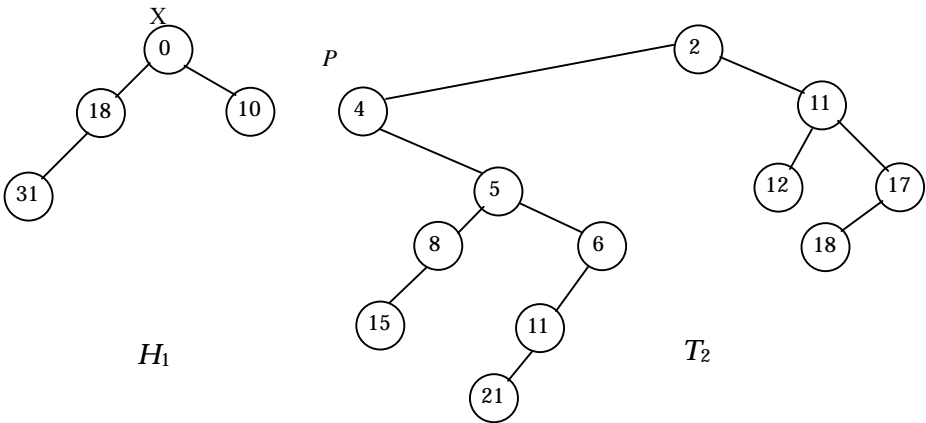



그림 11-12. 잘라 낸후 2개의 나무

이 두개의 나무들이 다 왼쪽더미이면 그것들은 $O(\log N)$시간에 병합되는데 그것을 실현해 보자. H_1이 왼쪽더미라는것을 쉽게 알수 있는데 왜냐하면 그것의 어느 매듭도 자식들의 어떤 변화를 가져 오지 않기때문이다. 그러므로 그것의 모든 매듭들이 왼쪽속성을 원래부터 만족시키므로 여전히 그 성질을 만족시켜야 한다.

그러나 이 계획은 실현할수 없는데 그 원인은 T_2가 반드시 제일 왼쪽이 아니기때문이다. 그러나 다음 두가지 관찰을 통하여 왼쪽더미의 속성을 쉽게 원상대로 회복한다.

- P로부터 T_2의 뿌리까지의 경로우의 매듭들만이 왼쪽더미속성을 위반할수 있는데

이것들은 자식들을 교체하여 고정할수 있다.

- 최대오른쪽 경로길이는 기껏해서 $\lfloor \log(N+1) \rfloor$매듭들을 가지므로 P로부터 T_2의 뿌리까지 경로우에서 첫 $\lfloor \log(N+1) \rfloor$매듭들만 검사하면 된다.

그림 11-13에서는 T_2가 왼쪽더미로 변환된후의 H_1과 T_2를 보여 주었다.

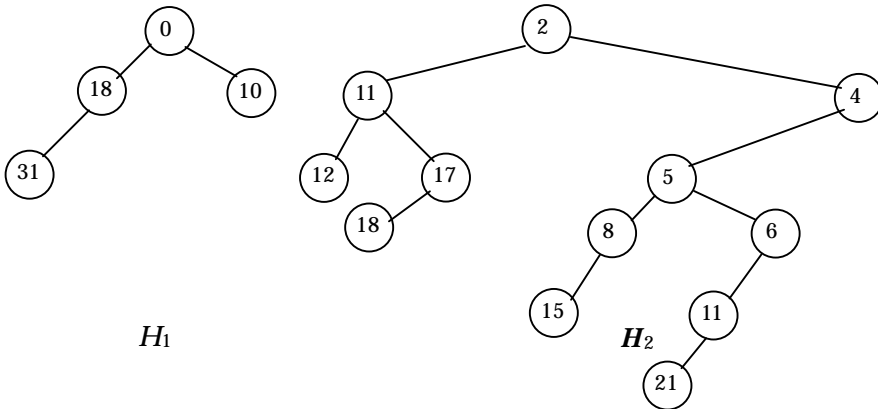



그림 11-13. 왼쪽더미 H_2로 변환된 T_2

$O(\log N)$걸음으로 T_2을 왼쪽더미 H_2에로 변환할수 있고 그다음 H_1과 H_2를 병합할수 있으므로 왼쪽더미에서 Decreasekey연산을 수행하는데 필요한 알고리듬은 $O(\log N)$시간이 걸린다. 실례에서 결과더미를 그림 11-14에 보여 주었다.

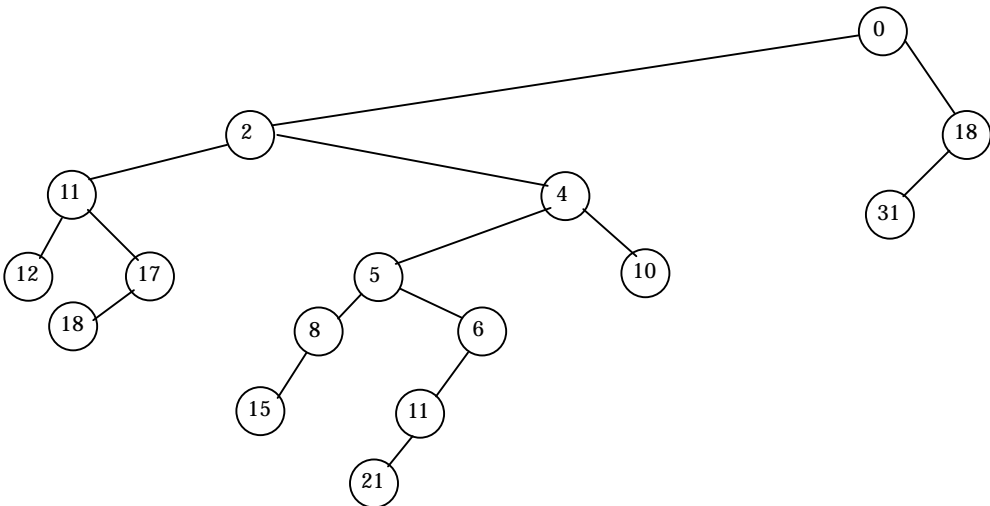



그림 11-14. H_1과 H_2를 련결하여 완성된 Decreasekey(X, 9)

2. 2항대기렬에 대한 지연병합

피보나치더미에서 리용되는 두번째 방법은 **지연병합(*lazy merging*)**이다. 이 방법을 2항대기렬에 적용하여 merge연산(특수한 경우인 삽입연산도 마찬가지이다.)을 수행하는데 걸리는 상환시간이 $O(1)$이라는것을 고찰하려고 한다. Deletemin연산에 대한 상환시간은 $O(\log N)$으로 된다.

방법은 다음과 같다. 2개의 2항대기렬을 병합하기 위하여 새로운 2항대기렬을 만들면서 2항나무들의 2개의 목록을 간단히 련결한다. 이 새로운 렬은 같은 크기의 나무들을 몇개 가질수 있으며 따라서 그것은 2항대기렬의 속성을 위반한다. 일관성을 위하여 이것을 **2항지연대기렬***(lazy binomial queue)*이라고 한다. 이 연산은 항상 상수시간(최악의 경우)을 가지는 빠른 연산이다. 앞에서 본것처럼 삽입연산은 한개 매듭의 2항대기렬을 만들고 병합하는 방법으로 진행된다. 차이는 merge가 **지연**연산이라는것이다.

Deletemin연산은 훨씬 더 어려운 연산인데 왜냐하면 2항대기렬을 최종적으로는 표준 2항대기렬로 다시 변환하지만 앞으로 보게 되는것처럼 그것이 여전히 $O(\log N)$의 상환시간을 취하기때문이다. 그러나 앞에서 본것처럼 최악의 경우의 시간은 $O(\log N)$이 아니다. Deletemin을 수행하기 위하여 최소원소를 탐색한다. 매개 자식들로 새 나무를 만들면서 앞에서처럼 대기렬로부터 그것을 제거한다. 그다음 2개의 같은 크기의 나무들을 불가능할 때까지 병합함으로써 이 모든 나무들을 2항대기렬로 병합한다.

실례로 그림 11-15에서는 지연2항대기렬을 보여 주었다. 지연2항대기렬에서는 하나 이상의 같은 크기의 나무가 존재할수 있다. Deletemin을 수행하기 위하여 앞에서처럼 최소의 원소를 제거하여 그림 11-16에 보여 준 나무를 얻는다.

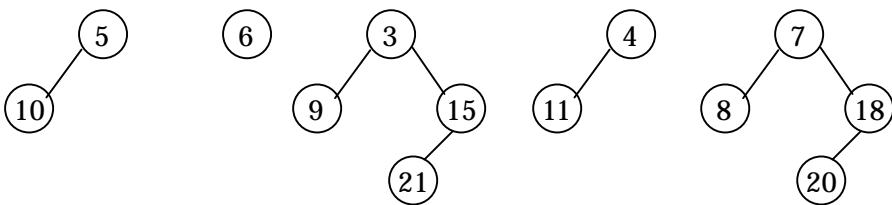



그림 11-15. 지연2항대기렬

이제는 이 모든 나무들을 병합하여 표준2항대기렬을 얻는다. 표준2항대기렬은 기껏해서 매개 위수에서 한개 나무를 가진다. 이것을 효률적으로 진행하기 위하여서는 merge연산을 주어 진 나무들의 수(T)에 비례하는 시간(혹은 항상 그보다 더 큰 ($\log N$))에 수행할수 있어야 한다. 이렇게 하기 위하여 목록 $L_{0,} L_1, \ldots, L_{R_{\max}+1}$의 묶음을 형성한다. 여기서 $R_{\max}$는 최대나무의 위수이다. 매개 목록 L_R는 위수가 R인 모든 나무들을 포함한다. 프

로그람 11-1에 보여 준 산법이 그다음에 적용된다.

```
/*1*/    for( R = 0; R <= ⌊log N⌋;   R++ )
/*2*/        whi1e( |L_R|>=2 )
             {
/*3*/            Remove two trees from L_R ;
/*4*/            Merge the two trees into a new tree;
/*5*/            Add the new tree to L_R+1 ;
             }
```

프로그람 11-1. 2항대기렬을 복원하는 방법

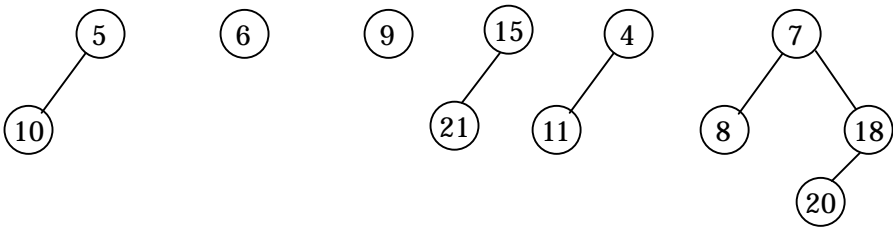



그림 11-16. 최소원소(3)를 제거한 이후의 지연 2항대기렬

순환이 매 회수에서 즉 3행부터 5행사이에서는 나무들의 총 개수가 1만큼 감소된다. 이것은 매번의 실행에서 일정한 시간이 걸리는 프로그람의 이 부분집행이 T-1시간동안 실행된다는것을 의미한다. 여기서 T는 나무들의 개수이다. For 순환이 반복되여 while순환의 끝에서의 시험들은 $O(\log N)$시간을 취하며 그래서 이 실행시간은 기대하던바대로 $O(T+\log N)$이다. 그림 11-17은 앞에서 보여 준 2항나무들의 모임에 이 산법을 실행한 결과를 보여 주었다.

지연2항대기렬의 상환분석

지연**2항대기렬의 상환분석**(*Amortized analysis of Lazy Binomial Queues*)을 진행하기 위하여 표준2항대기렬에 리용되였던것과 꼭같은 포텐샬함수를 리용한다. 그러므로 지연2항대기렬의 포텐샬은 나무들의 개수이다.

정리 11-3.

지연2항대기렬에 대한 merge와 insert연산들의 상환된 실행시간은 둘다 $O(1)$이다. Deletemin의 상환된 실행시간은 $O(\log N)$이다.

증명:

포텐샬함수는 2항대기렬의 모임안의 나무들의 개수이다. 초기의 포텐샬은 0이며 포텐샬은 항상 부수가 아니다. 그러므로 연산들의 렬에서 총 상환시간은 전체 실제시간에 대한 상환이다.

Merge연산인 경우 실제시간은 상수이며 2항대기렬들의 모임에서 나무들의 개수는 변하지 않는다. 따라서 식 11-2에 의해 상환된 시간은 $O(1)$이다.

Insert연산인 경우 실제시간은 상수이며 나무들의 개수는 기껏해서 1만큼씩 증가할수 있다. 그러므로 상환된 시간은 $O(1)$이다.

Deletemin연산은 좀 더 복잡하다. R가 최소원소를 포함하는 나무의 위수라고 하고 T를 나무들이 개수라고 하자. 그러면 deletemin연산의 시작에서는 포텐샬이 T로 된다. Deletemin연산을 수행하기 위하여 최소매듭의 자식들은 개개의 나무들로 가른다. 이것은 $T+R$개의 나무들을 만드는데 이 나무들은 표준2항대기렬로 병합되여야 한다.

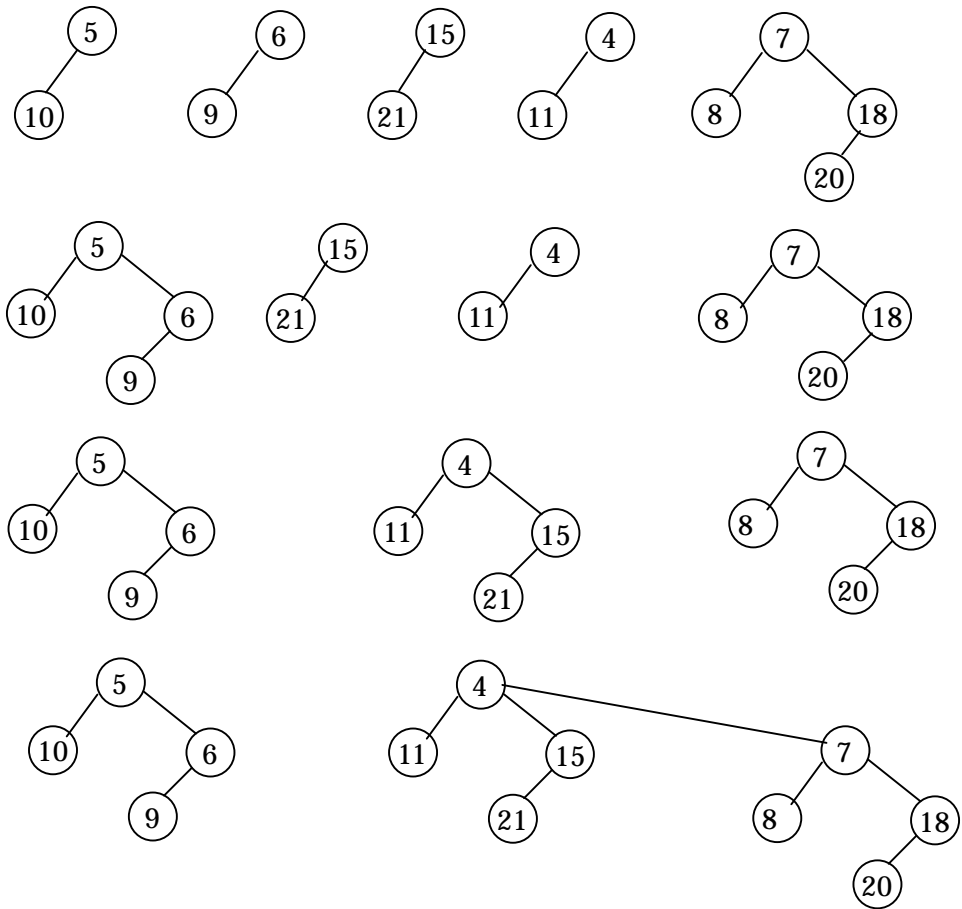



그림 11-17. 2항나무를 2항대기렬과 결합한다.

이것을 병합하는데 걸리는 실제시간은 앞에서[31]의 증명에 의하여 큰O표기법의 상수를 무시한다면 $T+R+\log N$으로 된다. 한편 일단 이것이 진행되면 기껏해서 $\log N$개의 나무들이 남아 있을수 있으며 그러므로 포텐샬함수는 기껏해서 $(\log N)-T$만큼 증가할수 있다. 포텐샬에서의 변화와 실제시간의 증가는 상환한계를 $2\log N+R$로 되게 한다. 모든 나무들이 2항나무들이므로 $R \leqq \log N$이다. 따라서 deletemin연산의 상환된 시간한계는 $O(\log N)$이다.

3. 피보나치더미연산

앞에서 언급한바와 같이 피보나치더미는 왼쪽더미의 decreasekey연산과 지연2항대기렬의 **병합**연산을 결합한다. 일정한 수정을 하지 않으면 이 2개의 연산들을 리용할수 없다. 2항나무들안에서 임의의 자르기가 진행되면 결과적인 숲은 2항나무들의 모임으로 되지 않는다. 때문에 매개 나무의 위수가 기껏해서 $\lfloor \log N \rfloor$이라는 사실이 더이상 성립되지 않는다. 지연2항대기렬에서의 deletemin연산에 대한 상환한계가 $2\log N+R$로 되였기때문에 deletemin연산에 대한 한계를 $R=O(\log N)$으로 유지할 필요가 있다.

$R=O(\log N)$이 성립하도록 하기 위해 뿌리가 아닌 모든 매듭들에 다음의 규칙들을 적용한다.

- 처음으로 자식을 잃었을 때(자르기때문에) 매듭(뿌리가 아닌 매듭)을 표식한다.
- 표식된 매듭이 또 다른 자식을 잃으면 그것을 부모로부터 잘라 낸다. 이제 이 매듭은 분리된 나무의 뿌리로 되며 더이상 표식되지 않는다. 이것을 **계단식자르기**라고 하는데 그것은 이것들중의 몇개는 하나의 decreasekey연산에 출현할수 있기때문이다.

그림 11-18은 decreasekey연산이전의 피보나치더미안의 한개의 나무를 보여 주었다. 열쇠가 39인 매듭이 12로 변할 때 이 더미의 순서가 위반된다. 따라서 이 매듭은 부모로부터 잘리워 지며 새로운 나무의 뿌리로 된다. 33을 포함하는 매듭이 표식되였기때문에 이것은 두번째로 잃어 진 자식으로 되며 따라서 그것은 부모(10)로부터 잘리워 진다. 이제 10은 그것의 두번째를 잃어 버렸으므로 수자 5로부터 잘리운다. 이 처리는 여기에서 정지되는데 왜냐하면 5번이 표식되지 않았기때문이다. 이제 매듭 5가 표식된다. 그 결과를 그림 11-19에 보여 주었다.

표식되였던 매듭들인 10과 33이 더이상 표식되지 않았는데 왜냐하면 이제는 그것들이 뿌리매듭들이기때문이다. 이것은 시간한계에 대한 증명에서 아주 중요한 관찰로 된다.

---

31 우리가 이렇게 할수 있는것은 큰O표기에서 암시되는 상수를 포텐샬함수안에 넣을수 있으며 그래도 증명에서 필요되는 함수들은 소거할수 있기때문이다.

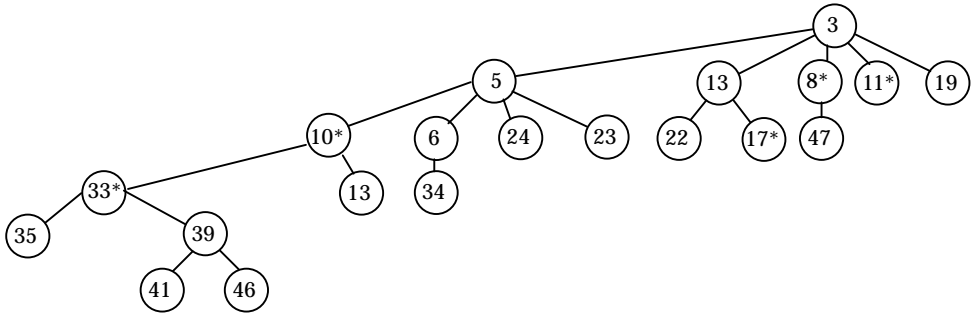



그림 11-18. 39를 12로 낮추기직전의 피보나치더미안의 나무

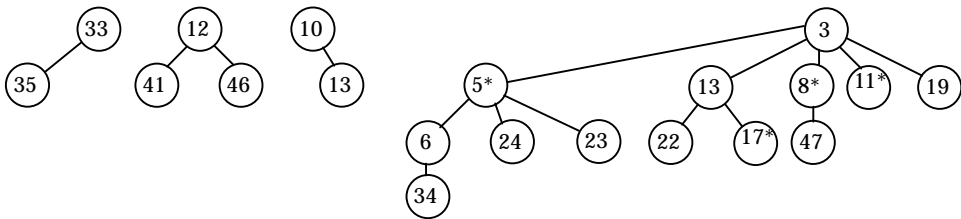



그림 11-19. decreasekey연산이 진행된후의 피보나치더미의 결과토막

4. 시간한계의 증명

매듭들을 표식하는 리유는 임의의 매듭의 위수 R(자식들의 수)를 한계지을 필요가 있기때문이라는것을 상기하자. 이제 N개의 자식들을 가지는 임의의 매듭이 위수 $O(\log N)$을 가진다는것을 보자.

보조정리 11-1.

X를 피보나치더미안의 임의의 매듭이고 c_i를 X의 i번째 가장 어린 자식이라고 가정하자. c_i의 위수는 적어도 $i-2$이다.

증명:

c_i가 X에 련결되였을 때 X는 이미(늙은)자식들 $c_1, c_2, \ldots, c_{i-1}$를 가지고 있었다. 그러므로 X는 c_i에 련결되였을 때 적어도 i-1개의 자식을 가진다. 매듭들이 같은 위수를 가질 때에만 련결되기때문에 c_i가 X에 련결될 때에는 적어도 i-1개의 자식들을 가지고 있다는것을 알수 있다. 이후에 기껏해서 한개의 자식매듭들을 잃을수 있거나 혹은 X로부터 잘리워 질것이다. 따라서 c_i는 적어도 i-2개의 자식들을 가진다.

보조정리 11-1로부터 위수가 R인 임의의 매듭은 많은 자식들을 가져야 한다는것을 쉽게 알수 있다.

보조정리 11-2.

F_k를 $F_0=1, F_1=1, F_k= F_{k-1}+ F_{k-2}$로 정의된(제1장 제2절에서) 피보나치수들이라고 하자. 위수가 $R \geqq 1$인 임의의 매듭은 적어도 F_{R+1}개의 자손(자기자체를 포함하여)들을 가진다.

증명:

S_R를 위수가 R인 최소나무라고 하자. 명백히 $S_0=1$이며 $S_1=2$이다. 따름 11-1에 의하여 위수가 R인 나무는 위수가 R-2, R-3,⋯, 1, 0인 부분나무들과 그리고 적어도 한개의 매듭을 가지는 단 하나의 부분나무를 가진다. S_R자체의 뿌리를 따라가 보면 이것은 $S_{R>1}$에 대하여 $S_R = 2+\sum_{i=0}^{R-2} S_i$의 최소값을 준다. $S_R = F_{R+1}$(련습문제 1-9 ㄱ)이라는 것을 쉽게 알수 있다.

피보나치수렬이 지수함수적으로 증가하기때문에 s개의 자식들을 가지는 임의의 매듭은 기껏해서 $O(\log s)$의 위수를 가진다는것을 쉽게 알수 있다.

보조정리 11-3.

피보나치더미안의 임의의 매듭의 위수는 $O(\log N)$이다.

증명:

우의 설명으로부터 직접 나온다.

우리가 목적하는 연산들 merge,insert,deletemin에 대한 시간한계를 계산해 내면 이것으로서 요구하는 상환시간한계를 증명할수 있다. 물론 피보나치더미들전체에서 중요한 문제는 decreasekey연산을 위해서도 $O(1)$시간한계를 가진다는것이다.

Decreasekey연산에 요구되는 실제시간은 연산과정에 수행된 종속적인 자르기의 회수에 1을 더한것과 같다. 계단식자르기회수는 $O(1)$보다 훨씬 더 클수 있기때문에 이에 대하여 포텐샬의 감소로써 보상해야 한다. 그림 11-19를 고찰하면 나무들의 개수는 매번의 계단식자르기에서 실제적으로 증가한다는것을 알수 있다. 따라서 계단식자르기과정에 감소되는것을 포텐샬함수로 확장해야 한다. 포텐샬함수로부터 나무들의 개수를 직접 상환해 낼수 없으므로 merge연산에 대한 시간한계를 증명할수 없다는것을 강조한다. 그림 11-19를 다시 고찰하면 종속적인 자르기는 표식된 마디의 위수를 감소시킨다. 왜냐하면

계단식자르기에 의해 희생되는 매개 매듭이 표식되지 않은 뿌리로 되기때문이다. 매개의 계단식자르기는 1단위의 실제시간을 소비하며 나무포텐샬을 1만큼 증가시키므로 매개의 표식된 매듭에 대하여 그 단위의 포텐샬을 고려하려고 한다. 이런 방법은 계단식자르기 수를 취소해 버릴 기회를 준다.

정리 11-4.

피보나치더미에서 Insert, merge, decreasekey연산들에 대한 상환된 시간한계는 $O(1)$이며 deletemin연산에 대해서는 $O(\log N)$이다.

증명:

포텐샬은 피보나치더미안의 나무들의 개수에 표식된 매듭들의 개수의 2배를 더한것이다. 역시 초기포텐샬은 0이며 항상 부수가 아니다. 따라서 연산렬에 거쳐서 총체적인 상환된 시간은 실제시간의 상한(웃한계)으로 된다.

Merge연산에서 실제시간은 상수이며 나무들의 수와 나무들에 표식된 매듭들의 개수는 변하지 않는다. 따라서 식 11-2에 의하여 상환된 시간은 $O(1)$이다.
Insert연산에서 실제시간은 상수이며 나무들의 개수는 1만큼씩 증가하며 표식된 매듭들의 개수는 변하지 않는다. 따라서 표텐샬은 기껏해서 1만큼씩 증가하며 상환된 시간은 $O(1)$이다. DeleteMin연산에서 R를 최소원소를 포함하는 나무의 위수라고 하고 T를 이 연산전의 나무들의 개수라고 하자. Deletemin을 수행하기 위하여 다시한번 나무의 자식들을 분리하여 추가적으로 R개의 새로운 나무들을 만들어 보자. 이것이 비록 표식된 매듭들을 제거할수 있어도(그것들을 표식되지 않은 뿌리들로 만들어 줌으로써) 이것은 임의의 추가적인 표식된 매듭를 만들어 낼수 없다는데 대하여 주의해야 한다.

이 R개의 새로운 나무들은 다른 T개의 나무들과 함께 정리 11-3에 의하여 $T+R+\log N = T+O(\log N)-T$의 시간안에 련결되여야 한다. 최대한 $O(\log N)$개의 나무들이 있을수 있고 표식된 매듭들의 개수는 증가할수 없기때문에 포텐샬변화는 최대한 $O(\log N)$이다. 실제시간과 포텐샬의 변화를 더하면 deleteMin연산의 상환된 한계 $O(\log N)$이 얻어 진다.

끝으로 decreaseKey연산에 대하여 C를 계단식자르기의 회수라고 하자. DecreaseKey의 실제시간은 $C+1$인데 이것은 수행된 자르기의 총 회수이다. 첫번째의(계단이 아닌)자르기는 새로운 나무를 만들며 따라서 포텐샬이 1만큼씩 증대된다. 매개의 계단식자르기는 새로운 나무를 만들지만 계단식자르기의 하나의 단위그물을 없애기 위해 표식되지 않은 (뿌리)매듭으로 표식된 매듭을 변환한다. 마지막자르기는 또한 표식안된 매듭(그림 11-19)을 표식된 매듭으로 변환하며 따라서 포텐샬을 2로 증가시킨다. 포텐샬에서 총 변화는 기껏해서 $3-C$이다. 더 나아가서 실제시간과 포

텐샬의 변화는 총적으로 4, 시간은 $O(1)$이다.

제5절. 펼친나무

마지막실례로 **펼친나무**(*Splay tree*)의 실행시간을 평가한다. 제4장으로부터 일부 항목 *X*를 호출한후 펼침단계는 *X*를 세개의 연산렬 왼쪽회전, 왼쪽-오른쪽회전, 왼쪽-왼쪽회전에 의하여 뿌리로 옮겨 진다는것을 상기하자. 이 나무회전을 그림 11-20에 보여 주었다. 나무회전이 매듭 *X*에서 수행된다면 회전에 앞서 *P*는 그의 부모(선조)이고 (만일 *X*가 뿌리의 자식이 아니라면) *G*는 그의 조부모라는것을 강조하게 된다.

매듭 *X*상에서 임의의 나무연산에 요구되는 시간은 뿌리로부터 *X*까지의 경로상에 있는 매듭들의 수에 비례한다는것을 상기하자. 만약 우리가 매개의 왼쪽회전연산을 한 회전으로 세고 매개의 왼쪽-왼쪽회전 혹은 왼쪽-오른쪽회전을 두회전으로 센다면 매개의 호출시간은 회전수에 1을 더한것과 같다.

펼침단계에서 최적화한계를 보여 주기 위하여 전체 펼침단계에서 기껏해서 $O(\log N)$까지 증가시킬수 있지만 또한 이 단계시에 집행되는 회전수를 취소하게 하는 포텐샬함수를 요구한다. 이 척도를 만족시키는 포텐샬함수를 구하는것은 쉽지 않다. 포텐샬함수에서 첫 추측은 나무에서 모든 매듭들의 깊이의 합일수도 있다는것이다.

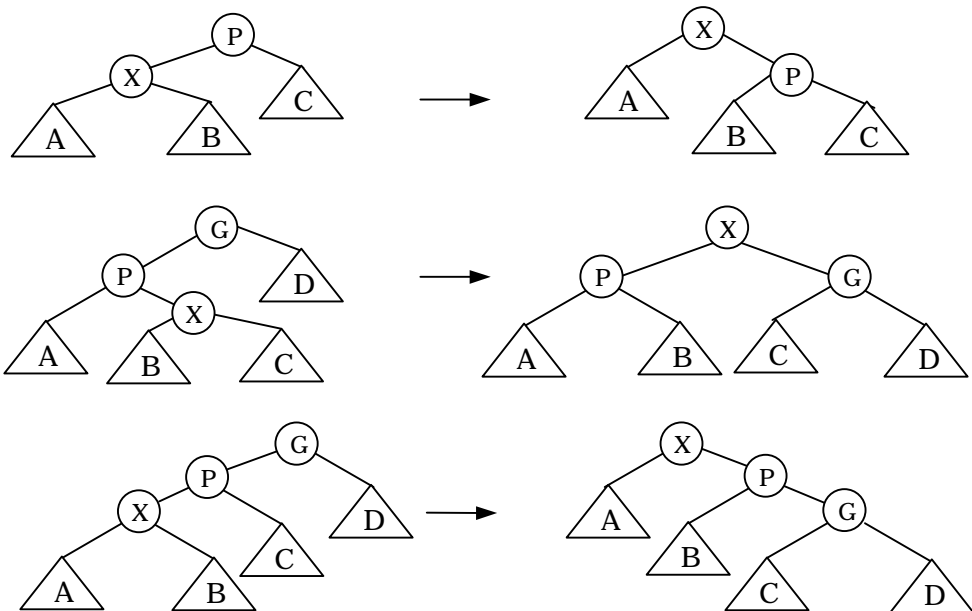



그림 11-20. 왼쪽회전, 왼쪽-오른쪽회전, 왼쪽- 왼쪽회전연산들 매개가 대칭인 경우

이것은 포텐샬이 호출하는 동안 $\Theta(N)$으로 증가될수 있기때문에 동작하지 않는다. 이에 대한 전형적인 실례는 원소들이 련속순서로 삽입될 때 발생한다.

포텐샬함수 Φ는

$$\Phi(T) = \sum_{i \in T} \log S(i)$$

로 정의된다.

여기서 $S(i)$는 (i자체를 포함하여) i의 자식들의 수를 표시한다. 포텐샬함수는 나무 T에서 모든 매듭 i에 대하여 $S(i)$의 로그합이다. 표시를 간단히 하기 위하여

$$R(i) = \log s(i)$$

로 정의한다.

이것은

$$\Phi(T) = \sum_{i \in T} R(i)$$

를 만든다.

$R(i)$는 매듭 i의 **위수**(*rank*)를 표시한다. 이 전용술어는 2항대기렬, 피보나치더미, *disjoint*설정알고리듬을 분석할 때 쓴것과 류사하다. 이 모든 자료구조들에서 *rank*의 의미는 조금씩 다르지만 *rank*는 일반적으로 나무의 크기의 로그적인 순서(크기)를 의미한다. N개의 매듭를 가진 나무 T에 대하여 뿌리의 위수(*rank*)는 간단히 $R(T)=\log N$이다. 위수들의 합을 포텐샬함수로 리용하는것은 높이의 합을 포텐샬함수를 리용하는것과 마찬가지이다. 중요한 차이는 회전이 나무에서 많은 매듭들의 높이를 변화시킬수 있는 한편 X, P, G만은 그것들의 변화된 위수를 가질수 있다는것이다.

기본정리를 증명하기전에 다음의 정리를 먼저 보자.

보조정리 11-4.

만약 $a + b \leqq c$이고 a와 b가 다 정수라면 $\log a + \log b \leqq 2\log c - 2$이다.

증명:

대수적 및 기하적인 평균부등식에 의하여

$$\sqrt{ab} \le (a+b)/2$$

따라서

$$\sqrt{ab} \le c/2$$

량변을 두제곱하면

$$ab \le c^2/4$$

로 된다. 량변에 로그를 취하면 정리는 증명된다.

예비적인 준비에 기초하여 기본정리를 증명할 준비가 되였다.

정리 11-5.

뿌리 T를 가진 나무를 펼치는데 걸리는 상환시간은 기껏해서 $3(R(T) - R(X)) + 1 = O(\log N)$이다.

증명:

포텐샬함수는 나무 T에서 매듭위수의 합이다. 만일 X가 T의 뿌리라면 회전은 없으며 포텐샬의 변화는 없다. 매듭을 호출하는데 걸린 실제시간은 1이며 따라서 최적시간도 1이며 정리는 옳다는것으로 증명된다. 그러므로 하나의 회전이 있다고 가정할수 있다.

임의의 펼침단계에 대하여 $R_i(X)$와 $S_i(X)$는 이 단계전의 X의 위수와 크기이고 $R_f(X)$와 $S_f(X)$는 펼침단계후의 즉시적인 X의 위수의 크기라고 하자. 왼쪽회전에 요구되는 상환시간은 $3(R_f(X) - R_i(X)) + 1$이며 왼쪽회전이나 왼쪽-왼쪽회전에 필요되는 상환시간은 기껏해서 $3(R_f(X) - R_i(X))$라는것을 보여 준다. 모든 단계들을 다 합할 때 그 합들은 설계시간한계까지사이에 있다.

왼쪽회전단계(*Zig-step*): 왼쪽회전단계에서 실제시간은 1(단일한 회전에 대하여)이며 포텐샬의 변화는 $R_f(X) + R_f(P) - R_i(X) - R_i(P)$이다. X와 P의 나무들만이 크기를 변화시키기때문에 포텐샬의 변화는 계산하기가 쉽다는것을 알수 있다. 따라서

$$AT_{zig} = 1 + R_f(X) + R_f(P) - R_i(X) - R_i(P)$$

그림 11-20으로부터 $S_i(P) \geqq S_f(P)$임을 알수 있으며 따라서 $R_i(P) \geqq R_f(P)$이다. 그러므로

$$AT_{zig} \leqq 1 + R_f(X) - R_i(X)$$

$S_f(X) \geqq S_i(X)$이므로 $R_f(X) - R_i(X) \geqq 0$이며 따라서 오른변을 증가시켜

$$AT_{zig} \leqq 1 + 3(R_f(X) - R_i(X))$$

를 얻을수 있다.

왼쪽-오른쪽회전단계(*Zig-Zag step*): 왼쪽-오른쪽회전경우에 대하여 실제시간계산량은 2이며 포텐샬의 변화는 $R_f(X) + R_f(P) + R_f(G) - R_f(X) - R_f(P) - R_f(G)$이다. 이것은 아래와 같은 상환된 시간한계를 준다.

$$AT_{zig\text{-}zag} = 2 + R_f(X) + R_f(P) + R_f(G) - R_i(X) - R_i(P) - R_i(G)$$

그림 11-20에서 $S_f(X) = S_i(G)$이고 따라서 이 위수들은 모두 같아야 한다는것을 알수 있다. 그래서 다음의 식을 얻는다.

$$AT_{zig\text{-}zag} = 2 + R_f(P) + R_f(G) - R_i(X) - R_i(P)$$

또한 $S_i(P) \geqq S_i(X)$라는것을 알수 있다. 결과적으로 $R_i(X) \leqq R_i(P)$이다. 대입하면

$$AT_{zig\text{-}zag} \leqq 2 + R_f(P) + R_f(G) - 2R_i(X)$$

그림 11-20에서 $S_f(P) + S_f(G) \leqq S_f(X)$라는것을 알수 있다. 만일 정리 11-4를 적용하면

다음식을 얻는다.

$$\log S_f(P)+\log S_f(G) \leqq 2\log S_f(X)-2$$

위수의 정의에 의하여 이것은

$$R_f(P) + R_f(G) \leqq 2R_f(X)-2$$

로 된다. 이것을 대입하면

$$AT_{zig\text{-}zag} \leqq 2R_f(X) - 2 R_i(X)$$
$$\leqq 2(R_f(X) - R_i(X))$$

$R_f(X) \geqq R_i(X)$로부터

$$AT_{zig\text{-}zag} \leqq 3(R_f(X) - R_i(X))$$

를 얻을수 있다.

왼쪽-왼쪽회전단계(*Zig-Zig step*): 세번째 경우는 왼쪽-왼쪽회전인 경우이다. 이 경우의 증명은 왼쪽-오른쪽회전과 류사하다. 중요한 부등식은 $R_f(X)= R_i(G)$, $R_f(X) \geqq R_f(P)$, $R_i(X) \leqq R_i(P)$, $S_i(X)+S_f(G) \leqq S_f(X)$이다. 련습문제 11-8에서 좀 더 구체적인것을 알게 된다.

전체적인 펼침단계에서 상환된 시간은 매개의 펼침단계에서의 상환된 시간의 합이다. 그림 11-21은 매듭 2에서 펼치기가 집행되는것을 보여 주었다. $R_1(2)$, $R_2(2)$, $R_3(2)$, $R_4(2)$가 4개 나무에서 매개 매듭 2의 위수라고 하자. 첫 단계는 왼쪽-오른쪽회전인데 그 시간은 $3(R_2(2)- R_1(2))$이다. 두번째 단계는 왼쪽-왼쪽회전인데 그 비용은 $3(R_3(2)- R_2(2))$이다. 마지막단계는 왼쪽회전이며 $3(R_4(2)- R_3(2))+1$보다 크지 않은 시간을 가진다. 따라서 전체 시간은 $3(R_4(2)- R_1(2))+1$까지이다.

일반적으로 모든 회전들(그중 하나는 기껏해서 왼쪽회전일수 있다.)의 상환비용들을 더함으로써 매듭 X에서 펼침을 실현하는데 드는 전체 상환시간은 $3(R_f(T)- R_i(X))+1$이라는것을 알수 있다. 여기서 $R_i(X)$는 첫 펼침단계에서 X의 위수이며 $R_f(X)$는 마지막펼침단계우의 X의 위수이다. 마지막펼침단계에서 뿌리 X가 남기때문에 $3(R_f(T)- R_i(X))+1$의 상환제한을 받게 되며 이것은 $O(\log N)$이다.

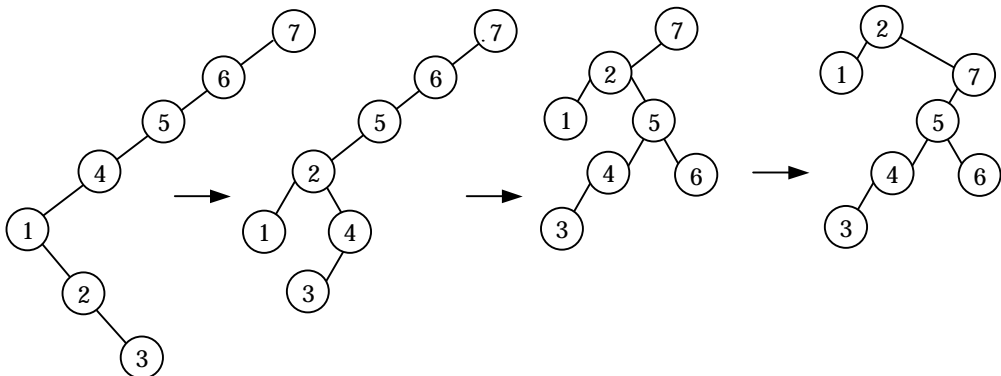



그림 11-21. 매듭에서 펼침에 포함한 단계들

펼친나무상에서 매 동작은 펼침을 요구하기때문에 어떤 동작의 알고리듬도 펼침의 상환시간의 요소를 안에 포함하고 있다. 따라서 모든 펼친나무동작들은 $O(\log N)$의 상환시간을 취한다. 보다 일반적인 포텐샬함수를 리용하여 펼친나무가 여러가지 주목할만한 속성들을 가진다는것을 보여 줄수 있다. 련습문제에서 이것을 구체적으로 취급한다.

요약

이 장에서 **상환분석**(*Amortized analysis*) 이 연산에 따르는 변화할당을 하는데 어떻게 리용될수 있는가를 보여 주었다. 분석을 실현하기 위하여 리상적인 포텐샬함수를 생각해낸다. 포텐샬함수는 체계의 상태를 측정한다. 고포텐샬(high potential)자료구조는 변덕스러우며 상대적으로는 보잘것 없는 조작들로 이루어 진다. 비용이 많이 드는 계산서가 어떤 조작에 의하여 발생할 때 그것은 이전의 조작들을 보관함으로써 지불된다. 비용이 매우 많이 드는 조작들은 자료구조가 고포텐샬을 가지고 할당된것보다 적은 상환시간을 리용했을 때에야만 일어 날수 있으므로 potential for disaster에 립각하여 포텐샬을 관찰할수 있다.

자료구조에서 저포텐샬은 매 조작시간이 그에 할당된 량과 대략 같다는것을 의미한다. 부의 포텐샬은 빚(debt)진다는것을 의미한다. 즉 배당된 시간보다 더 많은 시간을 소비하며 따라서 할당된(상환된) 시간은 한계가 없다.

식 11-2로 표현한것처럼 어떤 조작에 대한 상환된 시간은 실제시간과 포텐샬변화의 합과 같다. 전체 조작렬을 취해 보면 렬에서 상환시간은 전체 렬시간+포텐샬에서 그물의 변화와 같다. 이 그물변화가 명확하다면 상환한계는 실제시간소비에 대한 웃한계를 주며 의미를 가진다.

포텐샬함수를 고르는 열쇠는 최소의 포텐샬이 알고리듬이 시작될 때 발생한다는것을 담보하며 비용이 적게 드는 조작들에 대해서는 포텐샬이 증가되며 비용이 많이 드는 조작들에 대해서는 포텐샬이 감소된다는것이다. 초과되거나 단축된 시간이 포텐샬의 반대변화에 의하여 측정된다는것은 중요하다. 이것은 때로 실행하기보다는 말하기가 더 쉽다.

련습문제

11-1. 2항대기렬에 M개의 원소들을 련속적으로 삽입할 때 그 M시간단위보다 더 작게 취해 보시오.

11-2. $N=2^k-1$의 원소들이 2항대기렬에 있다고 하자. 교대적으로 M번의 insert와 deleteMin쌍연산을 진행하시오. 명백히 매 동작은 $O(\log N)$이 걸린다. 왜 이것은 삽입에 대하여 상환한계 $O(1)$을 반박하는가?

11-3. 본문에서 서술된 경사더미에서 상환한계$O(\log N)$는 merge에 요구되는 시간

(N)으로 상환되는 조작렬을 줌으로써 최악의 경우 한계로 전환될수 없다는 것을 설명하시오.

11-4. 하나의 top-down pass를 가진 두개의 경사더미들을 어떻게 합치며 상환시간 $O(1)$으로 merge시간을 얻어 낼수 있는가.

11-5. 상환시간 $O(\log N)$에서 조작을 지원하도록 경사더미를 확장시키시오.

11-6. 피보나치더미들을 실행시키고 그것을 딕스트라알고리듬을 리용할 때 2진더미집행과 비교하시오.

11-7. 피보나치더미의 표준집행은 매듭당 네개(부모, 자식, 두형제들)의 련결을 요구한다. 실행시간내에 기껏해서 일정한 인수의 비용에 한하여 련결수를 어떻게 줄이는가를 보여 주시오.

11-8. 왼쪽-오른쪽회전펼침의 상환시간은 기껏해서 $3(R_f(X)- R_i(X))$이라는것을 알아 내시오.

11-9. 포텐샬함수를 변화시켜서 펼침에 대한 여러가지 시간한계를 증명할수 있다. 무게함수 $W(i)$는 나무에서 매개 매듭에 련결하는 어떤 함수라고 하자. $S(i)$는 j를 포함하는 i의 뿌리내린 보조나무의 모든 매듭들의 무게의 합이라고 하자. 모든 매듭들에 대하여 특별한 경우 $W(i)=1$은 펼침한계를 증명하는데 리용된 함수에 관계된다. N을 나무에서 매듭들의 수라고 하고 M을 호출수라고 하자. 다음 두개의 정리를 증명하시오.

ㄱ. 전체 호출시간은 $O(M+(M+N)\log N)$

ㄴ. 만일 q_i가 항목 i를 호출하는 시간수이고 모든 i에 대하여 $q_i>0$이라면 호출시간의 총합은

$$O\left(M+\sum_{i=1}^{N} q_i \log(M/q_i)\right)$$

11-10. ㄱ. N개의 단일요소나무들에서부터 시작되여 $N-1$의 임의의 렬의 련결은 $O(N\log^2 N)$시간이 걸린다.

ㄴ. 한계를 $O(N\log N)$으로 개선하시오,

11-11. 제5장에서 rehashing을 서술하였다. 표가 절반이상 차게 될 때 2배만큼 큰 새 표가 구성되여 전체의 낡은 표는 바뀌여 진다. 삽입상환시간이 여전히 $O(1)$임을 보여 주기 위하여 포텐샬함수를 가지고 선형적인 상환분석을 집행하시오.

11-12. 만약 삭제가 허용되지 않는다면 N-매듭**2-3나무**(2-3 *tree*)에로 삽입되는 M개의 임의의 렬은 $O(M+N)$의 매듭분렬을 가져 온다는것을 증명하시오.

11-13. 더미순서화된 **량끝대기렬**(*Deque*)은 다음의 연산이 가능한 항목들의 목록을

포함하는 자료구조이다.

Push(x): 량끌대기렬의 끌앞에 항목을 삽입한다.

Pop(): 량끌대기렬로부터 앞의 항목을 제거하고 그것을 되돌려 준다.

Inject(x): 항목 X를 량끌대기렬의 끌뒤에 삽입한다.

Eject(): 량끌대기렬에서 뒤항목을 제거하고 그것을 되돌려 준다.

FindMin(): 량끌대기렬에서 제일 작은 항목(련결을 독단적으로 끊어 버린다.)을 되돌려 준다.

ㄱ. 매 조작당 일정한 상환시간안에 이 조작들을 하도록 하자면 어떻게 해야 하는가.

ㄴ. 매 조작마다 최악의 경우 일정한 시간내에 이 조작들을 하도록 하자면 어떻게 해야 하는가.

11-14. 2항대기렬이 상환시간 $O(1)$내에 실지로 결합될수 있는가. 나무들의 수와 가장 긴 나무의 위수의 합으로 정의되는 2항대기렬의 포텐샬을 정의하시오.

11-15. 시간을 절약하려는 시도에서 우리가 매개의 두번째 나무에 대하여 펼치기 조작을 하려고 한다. 상환비용이 로그적으로 계산되도록 해 보시오.

11-16. 펼친나무의 시간한계를 증명하는데서 포텐샬함수를 써서 펼친나무의 최대, 최소포텐샬은 무엇인가. 하나의 펼침에서 포텐샬함수는 얼마까지 감소할수 있는가. 하나의 펼침에서 포텐샬함수는 얼마까지 증가할수 있는가? 당신은 큰O표기법대답을 줄수 있다.

11-17. 펼침결과로써 호출하는 대다수의 매듭들은 뿌리를 향하여 중간으로 이동되며 한편 경로상의 매개 매듭쌍들은 한 준위 내려 간다. 이것은 매 매듭들의 길이의 로그값을 모든 매듭들에 걸쳐 합한것을 포텐샬함수로 리용한다는것을 립증한다.

ㄱ. 포텐샬함수의 최대값은 얼마인가.

ㄴ. 포텐샬함수의 최소값은 얼마인가.

ㄷ. ㄱ와 ㄴ에 대한 대답의 차이는 포텐샬함수가 그리 좋지 않아도 지적할 수 있다. 펼침조작을 $O(N\log N)$로 증가시킬수 있음을 증명하시오.

11-18. 피보나치더미의 최대깊이는 얼마인가?

참고문헌

상환분석의 구체적인 개괄을 문헌[10]에서 준다.

아래 참고서들의 대부분은 앞장들에서 인용되였다. 편리성과 안정성을 위하여 그것들을 다시 인용한다. 2항대기렬은 문헌[11]에서 처음으로 서술되였고 [1]에서 분석되였

다. 련습문제 11-3과 11-4의 풀이는 문헌 [9]에 있다. 피보나치더미는 [3]에서 서술한다. 련습문제 11-9 ㄱ는 가장 좋은 정적탐색나무들의 일정한 요소에 한하여 펼친나무가 최적임을 보여 주었다. 련습문제 11-9 ㄴ는 펼친나무가 가장 좋은 최적나무의 일정한 인수안에서 최적이라는것을 보여 주었다. 이것은 두개의 다른 결과들에서 잘 알수 있는것처럼 기본펼친나무들을 [7]에서 증명하였다.

펼친나무에 대한 merge조작은 [6]에 서술된다. 련습문제 11-12는 상환의 절대적인 리용을 통하여 [2]에서 푼다. 이 책에서는 또한 **2-3나무**(2-3 *tree)*들의 효과적인 련결방법을 보여 주었다. 련습문제 11-13에 대한 상환분석은 [4]에서 구해 준다. 련습문제 11-14는 [5]에서 준다. 상수요소가 비직결알고리듬보다 더 긴 경우 대기렬들을 조작하는 직결알고리듬을 설계하는 방법을 문헌[8]에서 보여 주었다.

1. M. R. Brown, "Implementation and Analysis of Binomial Queue Algorithms," *SIAM Journal on Computing.* 7 (1978), 298-319.
2. M. R. Brown and R. £. Tarian, "Design and Analysis of a Data Structure for Representing Sorted Lists," *SIAM Journal mi Computing* 9 (1980), 594-614.
3. M. 1.. Fredman and R. E. Tarian, "Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms," *Journal of the ACM,* 34 (1987), 596-615.
4. Ft. Gajewska and R. E. Tarjan, "Deques with Heap Order," *Information Processing letters,* 22 (1986), 197-200.
5. C. M. Khoong and H. W. Leong, "Double-Ended Binomial Queues," *Proceedings of the fourth Annual International Symposium on Algorithms and Computation* (1993), 128-137.
6. G. Port and A. Mortal, "A Fast Algorithm for Melding Splay Trees," *Proceedings of First Workshop on Algorithms and Data Structures* (1989), 450-459.
7. D. D. Sleator and R. E. Tarjan, "Self-adjusting Binary Search Trees," *Journal of the* ACM, 32 (1985), 652-686.
8. D. D. Sleator and R. E. Tarjan, "Amortized Efficiency of List Update and Paging Rules," *Communications of the ACM,* 28 (1985), 202-208.
9. D. D. Sleator and R. E. Tarjan, "Self-adjusting Heaps," *SIAM Journal on Computing,* 15 (1986), 52-69.
10. R. E. Tarjan, "Amortized Computational Complexity," *SIAM Journal on Algebraic and Discrete Methods,* 6 (1985), 306-318.
11. J. Vuillcmin, "A Data Structure for Manipulating Priority Queues," *Communications of theACM,*21(1978), 309-314.

제12장. 개선된 자료구조와 실현

이 장에서는 실천적인 측면에 중심을 둔 7가지 자료구조를 설명한다. 먼저 제4장에서 설명한 AVL나무를 대신할수 있는것들을 검토하는것으로부터 시작한다. 여기에는 펼친나무, 흑적나무, 건너뛰기목록(제10장에서 이미 고찰한)의 결정형태, A-A나무, 트리프에 대한 최적화된 내용들이 포함된다.

그다음 다차원자료에 리용할수 있는 자료구조들을 검토한다. 이 경우 매개 항목은 여러개의 열쇠를 가질수 있다. *k*차원나무는 여러개의 열쇠와 관계되는 탐색을 진행할수 있게 한다.

마지막으로 쌍더미를 검토하는데 이것은 피보나치더미를 제일 실천적으로 대신할수 있는것으로 되고 있다.

상기해야 할 항목들은 다음과 같다.

- 적합한 경우 비재귀적인 **내리탐색나무의 실현**(올리탐색대신에)
- 다른 매듭들중에서 감시매듭들을 리용하는 구체적이며 최적적인 실현

제1절. 내리펼친나무

제4장에서 **펼친나무**(*splay tree)*의 기본연산에 대하여 보았다. 항목 *X*가 잎으로 삽입될 때 펼치기라고 하는 나무에 대한 회전렬들은 *X*를 나무의 새 뿌리로 만든다. 또한 탐색이 진행되는 동안에 펼치기가 진행되는데 만일 탐색하려는 항목을 발견하지 못하면 접근경로의 마지막매듭에서 펼치기가 진행된다. 제11장에서 펼친나무연산의 유도시간이 $O(\log N)$이라는것을 보았다.

이 전략의 직접적인 실현은 뿌리로부터 나무의 아래쪽으로의 순회와 다음 펼치기단계를 실현하기 위한 올리순회를 요구한다. 이것은 부모와의 련결을 유지하거나 탄창에 접근경로를 기억시킴으로써 수행할수 있다. 그러나 이 두 방법들은 보충적인 처리를 많이 요구하며 여러가지 특수한 경우들을 취급하여야 한다. 이 절에서는 초기 접근경로에서 회전이 어떻게 수행되는가를 보게 된다. 그 결과 실제적으로 빠르고 $O(1)$의 보조공간만을 리용하지만 $O(\log N)$의 유도시간의 한계를 유지하는 하나의 절차가 얻어 진다.

그림 12-1에 왼쪽, 왼쪽-왼쪽, 왼쪽-오른쪽회전에 대한 경우를 보여 준다(일반적인 관례에 따라 대칭적인 세가지 경우에 대한 회전은 략한다.). 어떤 접근점에서 현재매듭 *X*는《중간》나무라고 부르는 그의 부분나무들의 뿌리이다.[32] 나무 *L*에는 T에서 *X*보다 작지만 *X*의 부분나무에는 없는 매듭들이 들어 있고 이와 류사하게 나무 *R*에는 T에서 *X*보

[32] 간단하게 고찰하기 위하여 ≪매듭≫과 그 매듭에서의 항목을 구별하지 않는다.

다 크지만 X의 부분나무에는 없는 매듭들이 들어 있다. 초기에 X는 T의 뿌리이며 L과 R는 비여 있다.

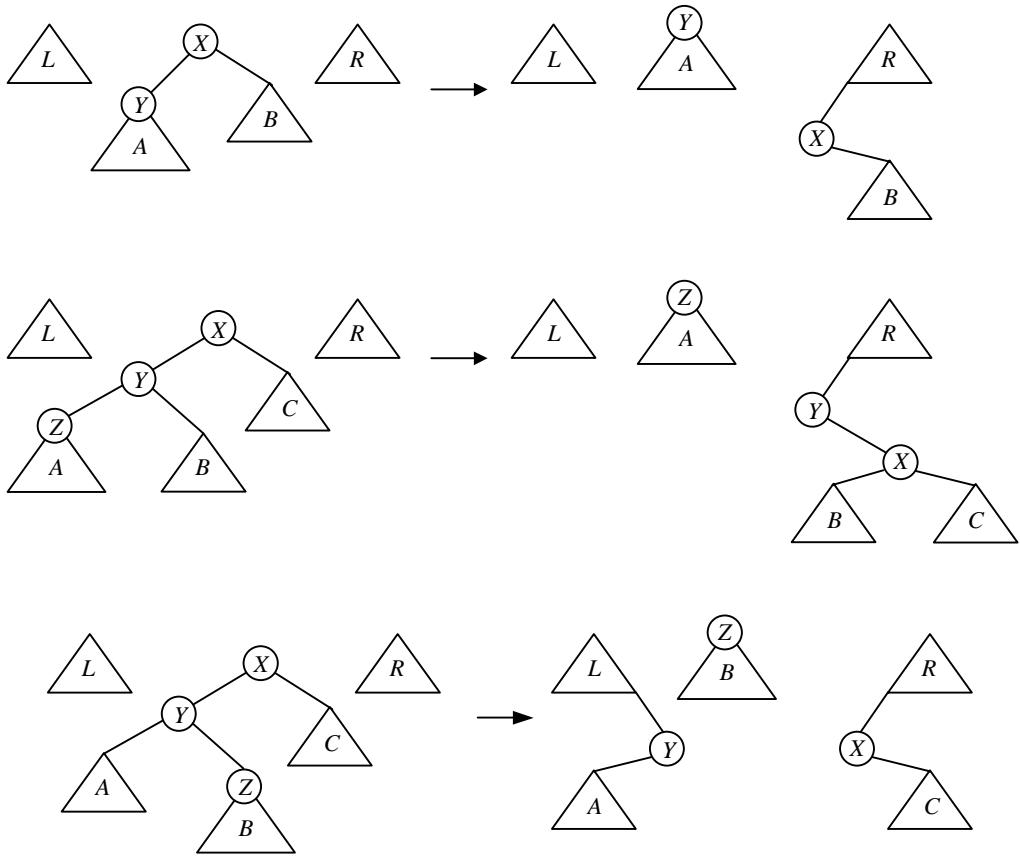



그림 12-1. 내리펼친나무의 회전들: 왼쪽, 왼쪽-왼쪽, 왼쪽-오른쪽

만약 회전이 왼쪽 방향으로 진행된다면 Y가 뿌리인 나무는 중간나무의 새로운 뿌리로 된다. X와 부분나무 B는 R의 가장 작은 항목의 왼쪽 자식으로 붙는다. 즉 X의 왼쪽자식은 론리적으로 NULL로 된다.[33] 결과 X는 R의 새로운 가장 작은 항목이다. 여기에서는 Y는 왼쪽회전인 경우에 잎으로 될수 없다는데 주의를 돌려야 한다. 만약 Y보다 더 작은 항목을 탐색하려 한다면 그리고 Y가 왼쪽 자식을 가지지 않는다면(그러나 오른쪽 자식은 가진다.) 왼쪽회전이 적용된다.

왼쪽-왼쪽회전에서도 이와 류사하게 고찰할수 있다. 중요한것은 회전이 X와 Y사이에서 진행된다는것이다. 왼쪽-오른쪽회전인 경우 맨 아래매듭 Z를 중간나무의 꼭대기로

[33] 코드에서 R 의 가장 작은 매듭은 필요 없으므로 NULL 인 왼쪽 련결은 가지지 않는다. 이것은 printTree®가 론리적으로 R 가 아닌 어떤 항목을 포함한다는것을 의미한다.

가져 가고 부분나무 *X*와 *Y*를 각각 *R*와 *L*에 붙인다. *Y*가 *L*에 붙은 다음 *L*에서 가장 큰 항목으로 된다는것을 명심하시오.

왼쪽-오른쪽회전단계는 회전이 진행되지 않기때문에 어느 정도 간단화할수 있다. *Z*를 중간나무의 뿌리로 만들 대신에 *Y*를 그 뿌리로 만든다. 이것을 그림 12-2에 보여 주었다. 이것은 왼쪽-오른쪽회전인 경우에 대한 작용이 왼쪽 경우와 같기때문에 코드를 더 간단히 할수 있다. 이것은 모든 경우들에 대한 시험 그자체가 곧 시간소비이므로 개선되였다고 볼수 있다. 결함은 다만 하나의 준위만 내려 감으로써 펼치기과정에 더 많은 반복이 있다는것이다.

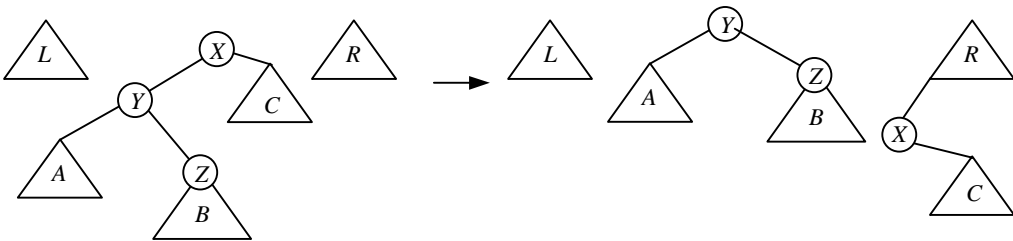



그림 12-2. 간단화된 내리왼쪽-오른쪽회전

그림 12-3은 마지막펼치기단계가 수행되였을 때 *L*, *R* 그리고 중간나무가 어떻게 단일한 나무형태로 정돈되는가를 보여 준다. 올리펼치기와 결과가 차이난다는데 주의를 돌리시오. 여기서 중요한것은 유도한계 $O(\log N)$이 유지된다는것이다(련습문제 12-1). 내리펼치기알고리듬의 실례를 그림 12-4에 보여 주었다. 이 나무에서 19에 접근하려고 시도한다고 하자. 첫 단계는 왼쪽-오른쪽회전이다. 그림 12-2(그림의 대칭판)와 같이 뿌리가 25인 부분나무를 중간나무뿌리로 하고 12와 그의 왼쪽 부분나무를 *L*에 붙인다.

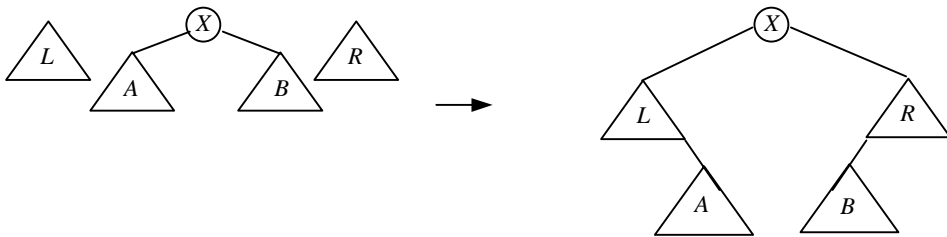



그림 12-3. 내리펼치기에 대한 마지막배렬

다음에 왼쪽-왼쪽회전 방식을 취한다. 15를 중간나무의 뿌리로 올리고 20과 25사이에 회전을 진행하면 결과 부분나무는 *R*에 붙는다. 그러면 19에 대한 탐색은 마지막왼쪽회전의 결과이다. 중간나무의 새로운 뿌리는 18이고 15와 그의 왼쪽 부분나무들은 *L*의 가장 큰 매듭의 오른쪽 자식으로 붙는다. 그림 12-3에 따라 다시 부분나무들을 모으기하

여 펼치기단계를 끝낸다.

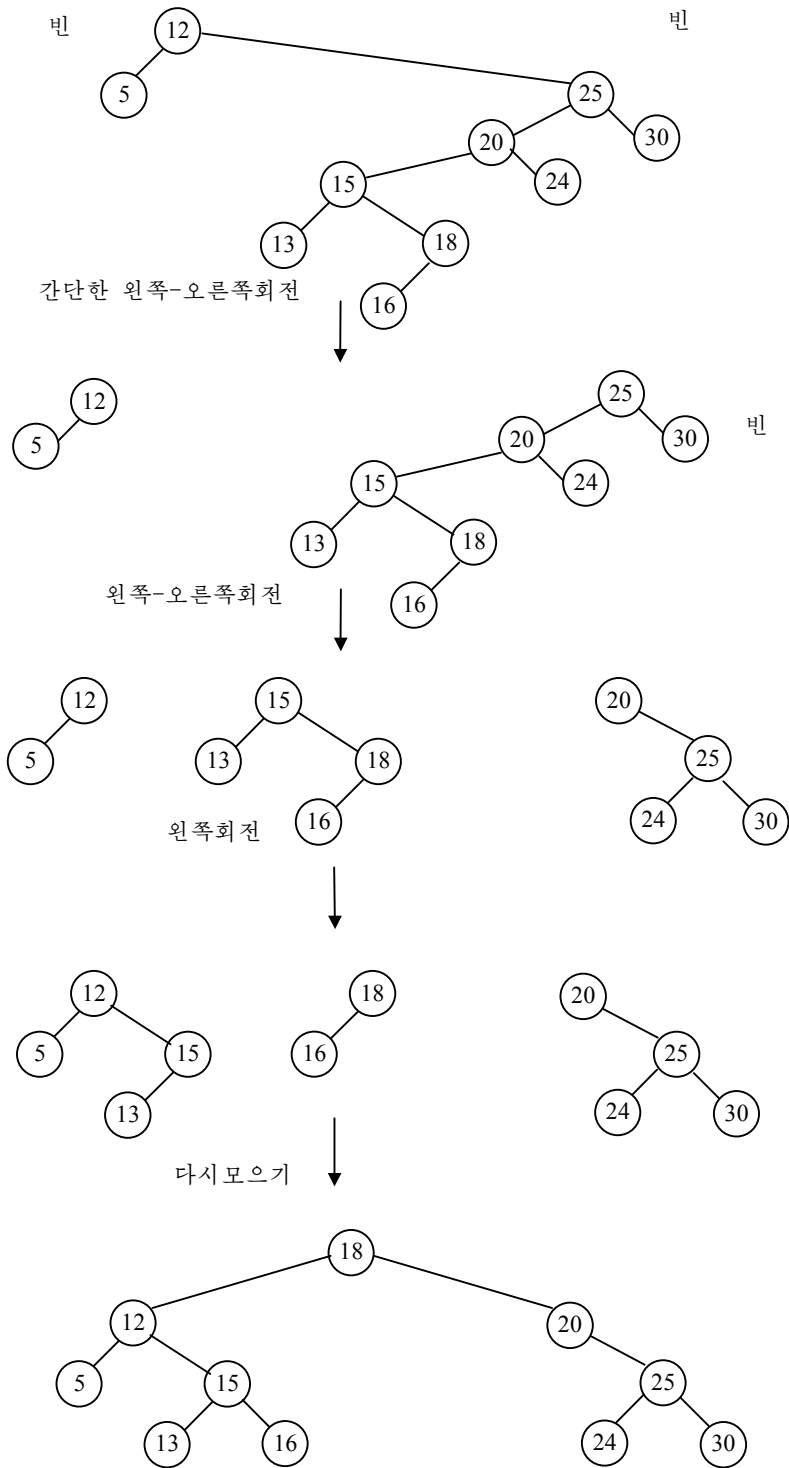



그림 12-4. 내리펼치기단계(나무에서 19에 접근하는 과정)

왼쪽과 오른쪽이 련결된 머리부를 리용하여 왼쪽 나무와 오른쪽 나무의 뿌리들을 포함시킨다. 이 나무들은 처음에 비여 있었기때문에 머리부는 이 초기상태에서 왼쪽 또는 오른쪽 나무의 최대 혹은 최소매듭에 각각 대응된다. 이 방법은 코드작성에서 빈 나무들의 검사를 피할수 있게 한다. 왼쪽 나무가 비지 않게 되는 첫 순간에 오른쪽 지적자는 초기화되며 더는 변하지 않는다. 이렇게 그것은 내리탐색끝에 오른쪽 나무의 뿌리를 가지게 된다. 이와 류사하게 왼쪽 지적자도 마감에 오른쪽 나무의 뿌리를 포함한다.

구축자와 해체자를 가진 펼친나무Tree클라스의 련결부를 프로그람 12-1에 보여 주었다.

```
template <c1ass Comparab1e>
class splay Tree
{
   public:
      explicit splay Tree( const Comparable & notFound );
      Sp1ayTree( const splay Tree & rhs );
      ~Sp1ayTree(    );
      const Comparable & f1ndMin(    );
      const Comparable & findMax(    );
      const Comparable & find( const Comparable & x );
      bool isEmpty(    ) const;
      void printTree(    ) const;
      Void makeEmpty(    );
      Void insert( const Comparable & x );
      Void remove( const Comparable & x );
      Const splayTree & operator=( const splayTree & rhs );
     Private:
      BinaryNode<Comparab1e> *root;
      BinaryNode<Comparab1e> *nu11Node;
      const Comparable ITEM_NOT_FOUND;
      const Comparable & elementAt( BinaryNode<Comparab1e> *t )const;
      void reclaimMemory( BinaryNode<Comparab1e> *t ) const;
      void printTree( BinaryNode<Comparab1e>    *t ) const;
      BinaryNode<Comparab1e>*    c1one( BinaryNode<Comparab1e> *t ) const;
              // Tree manipulations

      void rotateWithLeftChi1d( BinaryNode<Comparab1e>*    & k2 ) const;
      void rotateWithRightChi1d( BinaryNode<Comparab1e>*    & kl ) const;
      void sp1ay( const Comparable & x, BinaryNode<Comparab1e>* &)const;};
template <c1ass Comparab1e>
Sp1ayTree<Comparab1e>::Sp1ayTree( const Comparable & notFound )
   : ITEM_NOT_FOUND( notFound )
{   nullNode = new BinaryNode<Comparab1e>;
    nullNode->1eft = nu11Node->right = nullNode;
    nu11Node->e1ement = notFound;
    root = nullNode;}
template <c1ass Comparable >
```

```
Sp1ayTree<Comparab1e>::~Sp1ayTree( )
{    makeEmpty( );
       delete nullNode;        }
```

프로그람 12-1. 펼친나무들: 클라스대면부, 구축자, 해체자.

구축자는 nullNode감시표식을 할당한다. 감시표식 nullNode를 리용하여 NULL지시기를 론리적으로 표시한다. 해체자는 makeEmpty를 호출한후에 이것을 지워 버린다. 이 방법을 반복 리용하여 코드를 간단화한다(그리고 결과적으로 코드를 어느 정도 더 빠르게 작성한다.). 프로그람 12-2는 펼치기과정에 대한 코드를 준다. 머리부매듭은 R가 빌수도 있다는 우려없이 X를 R에서 가장 큰 매듭에 붙일수 있다는것을 확신하게 해준다(그리고 대칭인 경우에 대해서도 류사하게 L로 취급).

```
/* Internal method to perform a top-down splay.
 * The last accessed node becomes the new root.
 * x is the target item to splay around. t is the root of the subtree to splay. */
template <c1ass Comparab1e>
void SplayTree<Comparable>: : ( const Comparable & x,
                                   BinaryNode<Comparab1e> * & t ) const
{
 BinaryNode<Comparab1e>    *leftTreeMax,    *rightTreeMin;
    static BinaryNode<Comparab1e>    header;
     header.left = header.right = nullNode;
     leftTreeMax = rightTreeMin = &header;
    nu1lNode->e1ement = x;              // Guarantee a match
for(    ;    ;    )
        if( x < t->element )
        { if( x < t->left->element )
                rotateWithLeftChild( t );
                if( t->left == nullNode )
                   break;
               // Link Right
rightTreeMin->left = t;    rightTreeMin = t;    t = t->1eft;}
          else if( t->e1ement < x )
          { if( t->right->element < x )
                    RotateWithRightChild( t );
if( t->right == nullNode )
                 break;
               // Link Left
               leftTreeMax->right = t; leftTreeMax = t; t = t->right;}
        else
           break;
               leftTreeMax->right = t->left;    rightTreeMin->1eft = t->right;
     t->left = header.right;            t->right = header.left;            }
```

프로그람 12-2. 내리펼치기방법

우에서 언급한바와 같이 펼치기의 마지막에서 다시 모으기하기전에 header. Right와 header. left는 각각 *R*와 *L*의 뿌리들을 가리킨다. 작성된 코드는 비교적 간단하다.

프로그람 12-3에 항목을 나무에 삽입하는 방법을 보여 주었다. 필요하다면 새 매듭을 할당하고 만일 나무가 비여 있다면 한개 매듭을 가진 나무를 만든다. 그렇지 않으면 뿌리를 삽입된 값 x주위로 펼친다. 새로운 뿌리의 자료가 x와 같다면 중복이 있게 되는데 말하자면 x를 재삽입할대신 앞으로의 삽입을 위하여 newNode를 보존하고 즉시에 되돌아 간다. 새 뿌리가 x보다 더 큰 값을 포함한다면 새 뿌리의 오른쪽 부분나무는 newNode의 오른쪽 부분나무로 되며 뿌리의 왼쪽 부분나무는 newNode의 왼쪽 부분나무로 된다. root의 새 뿌리가 x보다 작은 값을 포함할 때에도 이와 류사한 론리를 적용한다. 이 두 경우에 newNode는 새 뿌리로 된다.

```
/*   Insert x   into the tree.     */
template <c1ass Comparab1e>
void Sp1ayTree<Comparab1e>: :insert( const Comparable & x )
{
  static BinaryNode<Comparab1e>   *newNode = NULL;
  if( newNode == NULL )
      newNode = new BinaryNode<Comparab1e>; newNode->e1ement = x;
   if( root == null Node )
   { newNode->1eft = newNode->right = nu11Node; root = newNode;    }
    else
   { sp1ay( x, root );
             if( x < root->e1ement )
             {   newNode->1eft = root->left; newNode->right = root;
              root->1eft = nullNode;       root = newNode;            }
              else
              if( root->e1ement < x )
              {   newNode->right = root->right;   newNode->1eft = root;
                 root->right = nu11Node;      root = newNode; }
             else
                 return; }
   newNode = NULL;     // So next   insert wi11 ca11 new
     }
```

프로그람 12-3. 내리펼친나무삽입

제4장에서 펼침이 뿌리에서의 지우기에 목적을 두기때문에 펼친나무에서 지우기는 쉽다는것을 보여 주었다. 프로그람 12-4에 지우기루틴을 보여 주는것으로서 이 과정을 끝마친다. 지우기공정이 대응하는 삽입공정보다 코드가 짧다는것은 명백한 사실이다. 프로그람 12-4는 또한 makeEmpty를 보여 준다. 나무매듭들을 개조하는 간단한 재귀적인

postorder의 순회는 펼친나무가 비록 좋은 동작을 준다해도 균형이 잘 맞지 않기때문에 불안전하다. 이 경우 재귀로 탄창기억구역이 초과될수 있다. 여전히 실행시간량이 $O(N)$ 인(비록 명백치 않아도)간단한 선택안을 리용한다. 연산자 =에 대해서도 류사하게 고찰하게 된다.

```
/* Remove x from the tree.  */
template <c1ass Comparab1e>
void Sp1ayTree<Comparab1e>: :remove( const Comparable & x )
{ BinaryNode<Comparab1 e>  *newTree;
    // If x is found, it will be at the root
    sp1ay( x, root );
    if( root->e1ement != x )
        return;     // Item not found; do nothing
    if( root->1eft == nullNode )
    newTree=root->right;
        else
  {   // Find the maximum in the left subtree。
        // Splay it to the root; and then attach right child
        newTree = root->left;    sp1ay( x, newTree );
        newTree->right = root->right; }
    delete root;    root = newTree;  }
/*  Make the tree logically empty.  */
 template<class Comparable>
 void
Sp1ayTree<Comparab1e>: :makeEmpty( )
{ findMax( );               // Splay max -item to root
    whi1e( !isEmpty( ) )
        remove( root->e1ement ); }
```

프로그람 12-4. 내리삭제공정과 makeEmpty

제2절. 흑적나무

력사적으로 **AVL나무**에 대응되는것이 **흑적나무**(*red-black*)이다. 흑적나무들에서의 연산들은 최악의 경우에 $O(\log N)$시간이 걸리며 아래에서 보게 되겠지만 삽입과 같은 비재귀적인 실현들은 AVL나무들에 비하여 헐하게 실현할수 있다.

흑적나무는 다음의 색특성을 가진 2진탐색나무이다.

① 매개 매듭은 붉은색 또는 검은색이다.

② 뿌리는 검은색이다.

③ 어떤 매듭이 붉은색이면 그의 자식은 검은색이여야 한다.

④ 한 매듭으로부터 NULL지적자까지의 매개 경로는 같은 수의 검은 매듭들을 포함하여야 한다.

채색규칙의 결과 흑적나무의 높이는 기껏해서 2log(N+1)이다. 결과 이러한 나무에 대한 탐색연산은 로그시간동안에 처리된다. 그림 12-5에 흑적나무를 보여 주었다. 붉은색매듭들은 2중원으로 표시된다.

보통 어려운것은 새로운 항목을 나무에 삽입하는것이다. 새 항목은 흔히 나무의 잎에 배치된다. 이 항목을 검게 색칠한다면 검은 매듭들의 경로가 더 길어 지기때문에 조건 ④를 위반하게 된다. 따라서 그 항목은 반드시 붉은색으로 칠해 져야 한다. 만일 부모가 검다면 수행이 가능하다. 부모가 이미 붉다면 붉은 매듭들을 련속적으로 가지게 되므로 조건 ③을 위반하게 된다. 이 경우에 조건 ③이 만족되도록 나무를 조정해야 한다(조건 ④에 위반되지 않게). 이것을 실현하는 기본조작은 색변환과 나무의 회전들이다.

1. 올리삽입

우에서 본것처럼 새롭게 삽입된 항목의 부모가 검은색이라면 문제가 없다. 그러면 그림 12-5에서 나무에 25를 삽입하는것은 그렇게 어렵지 않다.

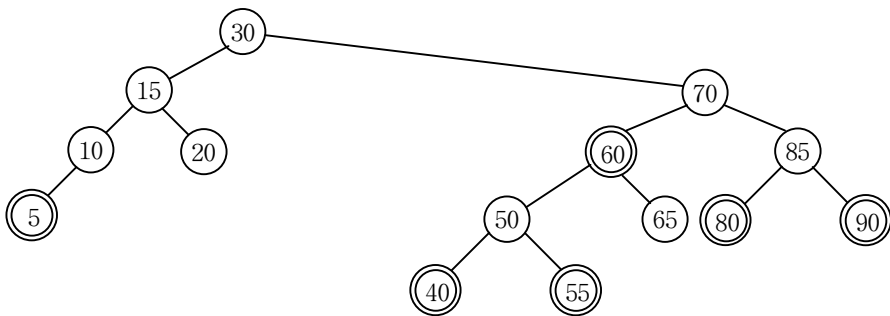



그림 12-5. 흑적나무의 실례(삽입서렬은 10, 85, 15, 70, 20, 60, 30, 50, 65, 80, 90, 40, 5, 55이다.)

만약 부모매듭이 붉다면 몇가지 경우들을 고찰하여야 한다(매개는 거울대칭을 가진다.). 먼저 그 부모매듭의 형제들이 검다고 가정하자(이것은 NULL매듭들이 검다는것을 의미한다.). 이것은 3 혹은 8을 삽입하는데 적용되지만 99를 삽입하는데는 적용될수 없다. X가 새롭게 잎에 추가되며 P는 그의 부모매듭이고 S는 부모의 형제매듭(만일 그것이 존재한다면), G는 조부모매듭이라고 하자. 다만 X와 P는 이 경우에 붉으며 G는 검은색이여야 한다. 왜냐하면 그렇지 않다면 삽입하기전에 두개의 련속적인 붉은 매듭들이 있는데

이것은 흑적나무의 규칙을 위반하는것으로 되기때문이다. 펼친나무에 대한 전문용어를 써서 *X*, *P*, *G*는 왼쪽-왼쪽련결 혹은 왼쪽-오른쪽련결(두 방향중의 어느 하나로)로 할수 있다. 그림 12-6은 이 경우에 나무회전을 어떻게 할수 있는가를 보여 주는데 여기서 *P*는 왼쪽 자식이다(대칭적인 경우이라는데 주의). 지어 *X*가 잎이더라도 나무의 중간에 위치하도록 하는 보다 일반적인 경우를 보여 주었다. 후에 이보다 일반적인 회전을 리용하게 될것이다.

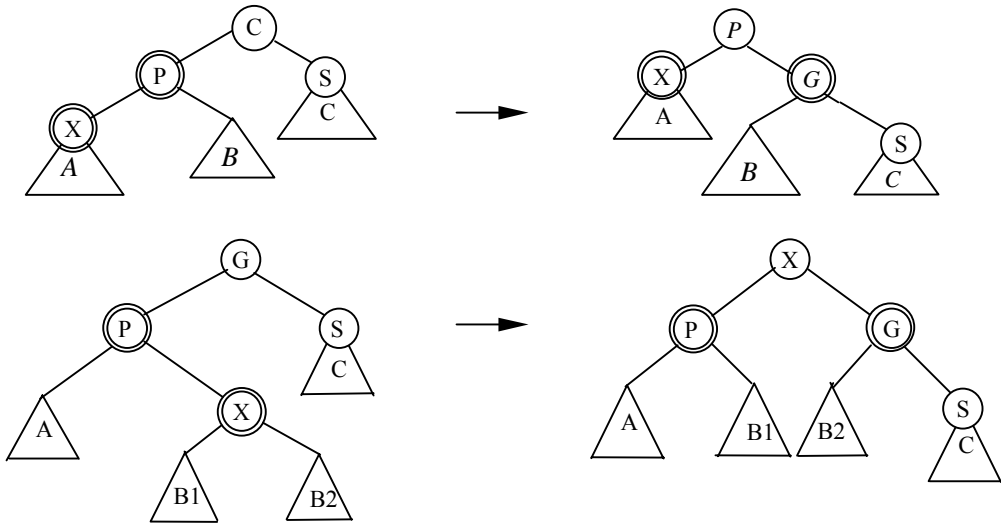



그림 12-6. S가 검을 때 왼쪽회전과 왼쪽-오른쪽회전

첫번째 경우는 *P*와 *G*사이에 단일회전에 대응한것이며 두번째 경우는 2중회전에 대응한것인데 처음에는 *X*와 *P*사이, 다음에는 *X*와 *G*사이에서 회전이 진행된다. 코드를 작성할 때 부모매듭과 또한 조부모매듭의 자리길을 보존해야 하며 그리고 다시 합치기를 위하여 증조부모매듭의 자리길도 보존해야 한다.

두경우 다 부분나무의 새 뿌리는 검은색으로 표시하며 그래서 원래의 증조부모가 붉은색이라고 하더라도 2개의 련속적인 붉은색매듭들이 있게 될 가능성은 제거된다. A, B, C통로우에 있는 검은색매듭들의 수는 회전결과 여전히 변하지 않고 그대로 남아 있다.

그런데 그림 12-5의 나무에서 79를 삽입하려는 경우와 같이 *S*가 붉은색이라면 어떻게 되겠는가. 이 경우 초기에는 하나의 검은 매듭이 부분나무뿌리로부터 *C*로 가는 경로상에 있다. 회전후에는 여전히 하나의 검은 매듭만이 있어야 한다. 그러나 두 경우에 *C* 경로상에는 세개의 매듭(새로운 뿌리, *G* 그리고 *S*)들이 있다. 이로부터 하나만이 검은색으로 되고 그리고 련속적인 붉은 매듭들을 가질수 없기때문에 이것은 *S*와 부분나무의 새 뿌리는 붉은색으로 표시하며 *G*는 검은색이여야 한다. 그런데 만약 증조부모매듭이 붉다

면 어떻게 되겠는가. 이 경우에 B나무와 2진더미들에서처럼 련속적인 두개의 붉은 매듭들을 련속적으로 가지지 않을 때까지 즉 뿌리에 도착(이것은 검은색으로 다시 칠할것이다.)할 때까지 뿌리를 향하여 우로 가면서 이 처리를 진행할수 있다.

2. 내리흑적나무

나무를 훑어 나갈 때 탄창 또는 부모련결을 리용하여 경로를 기억시켜야 할것이다. 이미 앞에서 내리절차를 쓰면 펼친나무들은 더 큰 효력을 가진다는데 대하여 고찰하였으며 이것은 S는 붉은색이 되지 말아야 한다는 담보를 주는 흑적나무에 내리방법을 적용할수 있다는것을 결론내릴수 있다.

그 방법은 개념적으로는 쉽다. 두개의 붉은 자식매듭들을 가지는 매듭 X가 있을 때 X를 붉게 하고 두개의 자식은 검은색으로 한다. 그림 12-7은 색번지기하는 과정을 보여준다. 이것은 X의 부모 P가 붉은색일 때에만 흑적나무의 규칙을 위반하는것으로 될것이다. 그러나 이 경우에 그림 12-6에서 적합한 회전을 적용할수 있다. X의 부모매듭의 형제가 붉은색이 될수 있는가. 이 가능성은 아래방향에 대한 조작에 의하여 제거되게 되는데 이로부터 X의 부모매듭의 형제는 붉은색일수 없다. 명백하게 나무아래방향에서 두개의 붉은 자식을 가지는 매듭 Y가 있으면 Y의 손자도 검은색이여야 하며 Y의 손자를 검은색으로 만들었기때문에 지어 회전후에도 두 준위에서 다른 붉은 매듭은 볼수 없을것이다. 그러므로 X에서 X의 부모가 붉은 색이면 X의 부모형제들에 대해 붉게 하는것은 불가능하다.

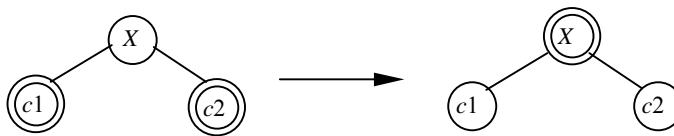



그림 12-7. 색번지기: X의 부모매듭이 붉은색일 때 회전을 계속한다.

실례에서 그림 12-5의 나무에 45를 삽입하려고 한다고 하자. 나무아래방향에서 두개의 붉은자식을 가진 매듭 50을 보게 된다. 그래서 색변환을 진행하여 50을 붉게 만들고 40과 55를 검게 한다. 그러면 50과 60은 둘다 붉은색이다. 60과 70사이에 단순한 회전을 하여 60을 30의 오른쪽 부분나무의 검은색뿌리로 70과 50을 둘다 붉은색으로 만든다. 다음 경로상에서 두개의 붉은 자식이 포함되는 다른 매듭을 보면 동일한 조작을 반복수행한다. 잎에 도달하면 삽입된 45는 붉은 매듭으로 부모는 검은색매듭으로 되며 수행은 끝난다. 결과적인 나무를 그림 12-8에 보여 주었다.

그림 12-8에서 보는것처럼 흑적나무는 주기적으로 잘 균형 잡힌 결과를 주는 나무이다. 실천적으로 증명해 보면 흑적나무의 평균깊이는 AVL나무의 평균깊이만큼 깊으며 또한 탐색시간은 최적값에 가깝다. 흑적나무의 우점은 삽입을 실현하는데 요구되는 시간한계가 상대적으로 낮으며 실지 회전들은 상대적으로 드물게 일어 난다.

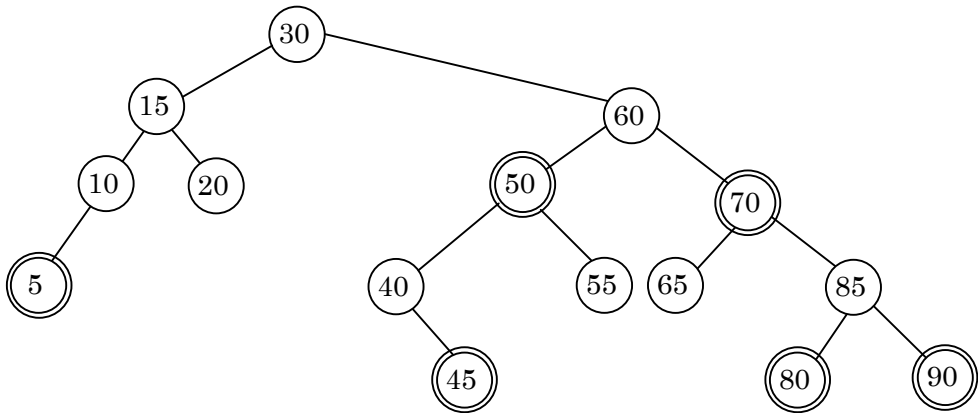



그림 12-8. 그림 12-5 에 45 의 삽입

실제집행은 가능한 회전이 많을뿐아니라 또한 일부 부분나무(오른쪽 부분나무 10과 같이)가 비여 있을수 있다는 가능성으로 하여 좀 복잡하다. 그리고 특이한 경우 즉 뿌리를 가지고 분리되는 경우(이때는 부모가 없다.)도 마찬가지이다. 그래서 두개의 감시표식매듭을 리용하는데 하나는 뿌리를 위한것이며 다른 하나는 펼친나무에서와 같이 NULL지시기를 지시하기 위한 nullNode이다. 뿌리감시매듭은 열쇠를 –∞로 보관하며 오른쪽은 실제뿌리에 련결한다. 이로부터 탐색과 출력처리들을 조정하여야 한다. 이것을 실현하는 재귀루틴은 아주 기교적이다. 프로그람 12-5에서는 나무의 뿌리순회과정을 어떻게 수정하는가를 보여 준다. printTree루틴들은 정확하다. t!= t->left에 대한 검사는 t!= nullNode로 서술할수 있다. 그러나 깊이를 복사하는 간단한 루틴에는 결함이 있다. 이것을 프로그람 12-5에 보여 주었다. 연산자 =는 별명을 검사하고 검사표식나무를 비게 한 후 clone을 호출한다. 그러나 clone에서 t==nullNode를 검사하지 않는데 이것은 nullNode가 목적하는 nullNode이며 원천nullNode가 아니기때문이다. 이렇게 더 기교 있는 검사를 진행한다.

프로그람 12-6에서는 구축자를 가진 RedBlackTree의 골격도을 보여 준다.

다음 프로그람 12-7은 단일회전을 수행하는 루틴을 보여 준다. 결과적인 나무가 부모에 붙여 져야 하므로 rotate는 인수로 부모매듭을 취하게 된다. 나무가 낮아 지는데 따라 회전의 형태를 추적해 가기보다는 오히려 항목을 인수로 주는 편이 더 낫다.

```
/**
 *Print the tree contents    in sorted order; calls Internal recursive printTree below.
 */
template <c1ass Comparab1e>
void RedB1ackTree<Comparab1e>::printTree( ) const
{
        if( isEmpty( ) )
            cout « "Empty tree" « endl;
        else
            printTree( header->right );
}
template <c1ass Comparab1e>
void RedB1ackTree<Comparab1e>::printTree( RedB1ackNode<Comparab1e> *t )    const
{
    if( t != t->1eft )
    {
        printTree( t->1eft );
        cout « t->e1ement « endl;
        printTree( t->right );
    }
}
/**
 * Deep copy; calls internal recursive clone below.
 */
template <c1ass Comparab1e> const
RedB1ackTree<Comparab1e> &
RedB1ackTree<Comparab1e>::operator=( const RedB1ackTree<Comparab1e> & rhs )
{
    if( this != &rhs )
    {
        makeEmpty( );
        header->right = c1one( rhs.header->right );
    }

    return *this;
}
template <c1ass Comparab1e>
RedB1ackNode<Comparab1e> *
RedB1ackTree<Comparab1e>: :c1one( RedBlackNode<Comparab1e>* t ) const
{
    if( t == t->1eft )              // Cannot test against nullNode!!!
        return nullNode;
    Else
        return new RedB1ackNode<Comparab1e>( t->e1ement, c1one( t->1eft ),
                                            c1one( t->right ), t->co1or );
}
```

프로그램 12-5. 두개의 표식을 가진 나무순회:printTree 와 연산자 =

```
template <c1ass Comparab1e>
class RedBlackNode
{
    Comparable       element;
    RedBlackNode     *1eft;
    RedBlackNode     *right;
    int              color;
    RedBlackNode( const Comparable & theElement = Comparable(    ),
                  RedBlackNode *lt = NULL, RedBlackNode *rt = NULL,
                  int c = RedB1ackTree<Comparab1e>::BLACK )
      : e1ement( theElement ), 1eft( 1t ), right( rt ), co1or( c ) { }
    friend class RedB1ackTree<Comparab1e>;
  };

  template <c1ass Comparab1e>
  class RedBlackTree
  {
     public
      explicit RedB1ackTree( const Comparable & negInf );
      RedBlackTree( const RedBlackTree & rhs );
      ~RedB1ackTree( );

      enum { RED, BLACK };

           // Usual public member functions (not shown)
    private:
      RedBlackNode<Comparab1e>    *header;     // The tree header (contains negInf)
      const Comparable ITEM_NOT_FOUND;
       RedBlackNode<Comparab1e>    *nullNode;

                  // Used in insert routine and its helpers (logically static)
      RedBl ackNode<Comparab1e>    *current;
      RedBl ackNode<Comparab1e>    *parent;
      RedBl ackNode<Comparab1e>    *grand;
      RedBl ackNode<Comparab1e>    *great;

           // Red-black tree manipulations
      void hand1eReorient( const Comparable & item );
      RedB1ackNode<Comparab1e>* rotate( const Comparable & item,
                                  RedB1ackNode<Comparable> *parent ) const;
           // Additional private member functions (not shown)
     }
/**
  * Construct the tree.
  * negInf is a value less than or equal to a11 others.
  * It is also used as ITEM_NOT_FOUND.
  */
  template <c1ass Comparab1e>
  RedBlackTree<Comparab1e>: :RedB1ackTree( const Comparable & negInf )
```

```
: ITEM_NOT_FOUND( negInf )
 {
        nullNode = new RedB1ackNode<Comparab1e>;
        nullNode->1eft = nu11Node->right = nullNode;
        header = new RedB1ackNode<Comparab1e>( negInf );
        header->left = header->right = nullNode;
 }
```

프로그람 12-6. 클라스련결부와 구축자

```
/**
* Internal routine that performs a single or double rotation.
* Because the result is attached to the parent, there are four cases.
* Called by handleReorient.
* item is the item in handleReorient.
* parent is the parent of the root of the rotated subtree.
* Return the root of the rotated subtree.
*/
template <c1ass Comparab1e>
RedBlackNode<Comparab1e>    *
RedBlackTree<Comparab1e>:: rotate( const Comparable & item,
                              RedBlackNode<Comparab1e> *theParent ) const
    {
        if( item < theParent->element )
        {
            item < theParent->right->element ?
                rotateWithLeftChild( theParent->right ) :          // LL
                rotateWithRightChild( theParent->right );          // LR
            return theParent->1eft;
        }
        else
        {
            item < theParent->right->element ?
                rotateWithLeftChild( theParent->right ) :          // RL
                rotateWithRightChild( theParent->right );          // RR
            return theParent->right;
            }
    }
```

프로그람 12-7. 회전방법

이로부터 삽입조작을 진행하는동안 회전수가 매우 적으리라고 기대하기때문에 이런 방법으로 하는것이 더 간단하고 빠르다는것을 알수 있다. Rotate함수는 적합한 단순회전 수행결과를 되돌려 준다.

마지막으로 프로그람 12-8에서 삽입방법을 보자. handleReorient루틴은 두개의 붉은

자식매듭을 가진 매듭과 만날 때와 또한 잎에 삽입할 때 호출된다. 2중회전은 실제로 단일회전을 두번 수행하는것으로 관찰해야 하며 또한 가장 재치 있는 기교부분은 X로 나가는 가지가 반대방향을 취할 때에만 실행된다는것이다. 이미 이야기한것처럼 insert는 나무가 낮아 질 때 부모, 조부모, 증조부모매듭들의 정보를 계속 받아야 한다. 주의할것은 회전후 조부모와 증조부모매듭에 기억된 값들은 더는 정확치 않다. 그러나 그것들은 다음에 필요될 때 회복될것이다.

```
/**
 * Internal routine that is called during an insertion if a node has two red
 * children. Performs flip and rotations, item is the item being inserted.
 */
template <c1ass Comparab1e>
void RedB1ackTree<Comparab1e>::hand1eReorient( const Comparable & item )
{
        // Do the color flip
    current->color = RED;
    current->left->co1or = current->right->co1or = BLACK;

    if( parent->co1or == RED )          // Have to rotate
    {
        grand->co1or = RED;
        if( item < grand->e1ement != item < parent->e1ement )
            parent = rotate( item, grand );     // Start dbl rotate
        current = rotate( item, great );    current->co1or = BLACK;
    }
    header->right->co1or = BLACK; // Make root black
}

/**
 * Insert item x into the tree. Does nothing if x already present.
 */
 template <c1ass Comparab1e>
 void RedBlackTree<Comparable>::insert( const Comparable & x )
 {
    current = parent = grand = header;
    nu11Node->e1ement = x;

    whi1e( current->e1ement != x )
    {
        Great = grand; grand = parent; parent = current;
        current = x < current->element ? current->left : current->right;

                // Check if two red children; fix if so
        if( current->1eft->co1or == RED && current->right->co1or == RED )
            hand1eReorient( x );
    }
```

```
        // Insertion fails if already present
    if( current != nullNode )
        return;
    current = new RedB1ackNode<Comparab1e>( x, nullNode, nullNode );

          // Attach to parent
    if( x < parent->element )
        parent->left = current;
    else
        parent->right = current;
    hand1eReorient( x );
}
```

프로그램 12-8. 삽입과정

3. 내리삭제

흑적나무에서 삭제는 또한 내리실현할수도 있다. 결국 이것을 리용하여 잎을 쉽게 삭제할수 있도록 한다. 그것은 두개의 자식을 가진 매듭의 삭제는 그것을 오른쪽 부분나무에서 가장 작은 매듭으로 치환하기때문이다. 이 매듭은 기껏해서 하나의 자식매듭을 가져야 하며 그렇게 되면 삭제된다. 하나의 오른쪽 자식만을 가진 매듭들도 같은 방법으로 삭제될수 있으며 한편 왼쪽 자식을 가진 매듭들은 왼쪽 부분나무에서 제일 큰 매듭과 치환함으로써만 삭제될수 있다. 주의해야 할것은 흑적나무에서 하나의 자식을 가진 매듭의 경우를 처리하는 전략은 허용하지 말아야 한다. 왜냐하면 그 경우 나무중간에서 두개의 붉은 매듭들을 련결함으로써 흑적나무에서 흑적조건의 실현을 어렵게 할수 있기때문이다.

물론 붉은색잎의 삭제는 어렵지 않다. 그러나 잎이 검다면 검은 매듭의 제거가 우에서 약속한 조건 4를 위반하는것으로 되기때문에 삭제는 좀 더 복잡해 진다. 해결책은 내리경로를 통과하는동안 잎이 붉은색이 되도록 하는것이다.

여기서 보는바와 같이 X는 현재매듭이고 T는 그의 형제이며 P는 그의 부모라고 하자. 뿌리를 검게 색칠하는것으로부터 시작한다. 나무아래로 내려 가면서 X가 붉은색이 되게 하려고 한다. 새 매듭에 도착했을 때 P는 붉으며 X와 T는 검다(두개의 련속적인 붉은 매듭들은 가질수 없기때문에)는것을 확인한다. 두가지 중요한 경우가 있다.

첫째로 X가 두개의 검은색자식을 가진다고 하자. 그러면 그림 12-9에서 보여 준것처럼 3개의 경우가 있다. 만일 T가 2개의 검은색자식을 가지면 X와 T ,P는 색번지기를 진행하여 변화되지 않도록 할수 있다. 한편 T의 자식중의 하나는 붉은색이다.

이에 준하여 그림 12-9의 두번째와 세번째 경우에 보여 준 회전을 적용할수 있다. 주의할것은 이 경우 nullNode는 검은색으로 고려되기때문에 잎에 적용해야 할것이다. 둘

째로 X의 자식중의 하나는 붉은색이라고 하자. 이 경우에 다음 준위에로 떨어 져 새로운 X와 T, P를 얻는다.

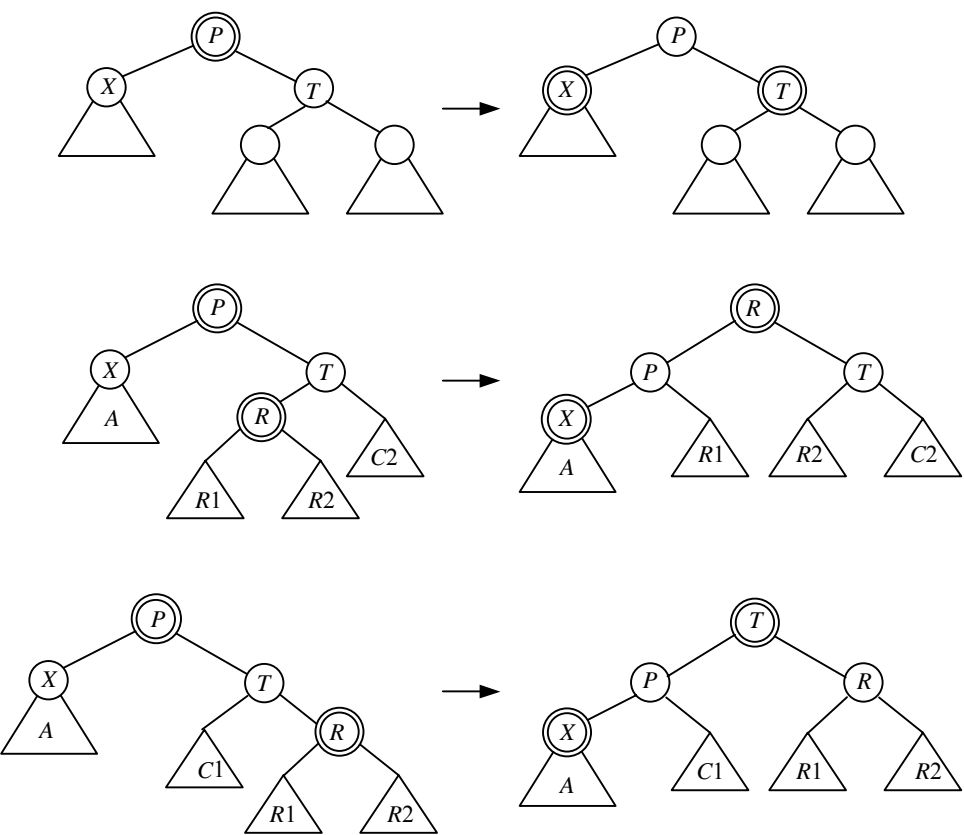



그림 12-9. X가 왼쪽 자식이고 두개의 검은 자식들을 가질 때의 세 경우

운수가 좋다면 X는 붉은색자식에 떨어 지며 그러면 우의 조작을 계속할수 있다. 만일 아니면 T는 붉게 될것이며 X와 P는 검게 된다. T와 P를 회전시켜 X의 새 부모매듭을 붉게 만들수 있다. 즉 X와 그의 조부모매듭은 물론 검게 될것이다. 이 점에서 첫번째 기본경우에로 되돌아 갈수 있다.

제3절. 결정성건너뛰기목록

흑적나무에서 리용한 고찰방식은 최악의 경우 로그적인 연산을 담보하는 건너뛰기목록에 적용할수 있다. 이 절에서는 자료구조를 형성하는 가장 간단한 실행방법인 1-2-3 **결정성건너뛰기목록**(*Deterministic Skip List*)을 보게 된다.

제10장으로부터 건너뛰기목록에서 매듭들은 임의로 부여 되는 높이를 가진다는것을 다시 상기한다. 높이가 h인 매듭은 h앞방향련결 p_1, p_2, $\cdots p_h$을 포함한다. p_i는 높이 i 혹은 그이상의 높이를 가진 다음 매듭에 련결한다. 매듭이 높이 h를 가질 확률은 0.5^h(0.5는 0과 1사이의 임의의 수로 치환될수 있다)이다. 결과적으로 준위가 아래로 하나 떨어 질때까지 다만 몇개의 앞방향련결처리가 요구된다. 이로부터 대략 $\log N$준위를 가지게 되며 연산당 기대되는 실행시간은 $O(\log N)$으로 된다.

이 한계를 최악의 경우 한계로 되도록 하자면 앞방향으로 련결하는 수를 낮은 준위로 내려 떨굴수 있을 때까지 시험하여야 한다. 이것을 수행하기 위하여 평형맞추기조건을 추가한다. 처음 두가지 정리가 요구된다.

정의: 한 요소에서 다른 요소에로 가는 적어도 하나의 련결이 존재한다면 두 요소들은 련결된다.

정의: h높이로 련결된 두 요소들사이 간격의 크기는 그것들사이에 있는 높이가 $h-1$인 요소수와 같다.

1-2-3결정성건너뛰기목록은 매개 간격(선두와 꼬리사이의 간격이 0으로 될 가능성은 제외)이 크기 1-2-3을 가지는 특성을 만족시킨다. 그림 12-10은 1-2-3결정성건너뛰기목록을 보여 준다. 크기가 3인 2개의 간격이 있다. 첫째것은 25와 45사이에 있는 높이가 1인 세개의 요소이고 두번째것은 목록의 선두와 꼬리사이에 높이가 2인 세개의 요소이다. 두번째것은 목록의 선두와 꼬리사이에 있는 높이가 2인 3개의 요소이다. 꼬리매듭은 무한대를 포함한다. 즉 이것의 존재로 알고리듬은 간단해 지며 목록의 끝에 있는 간격에 대한 정의를 쉽게 할수 있다.

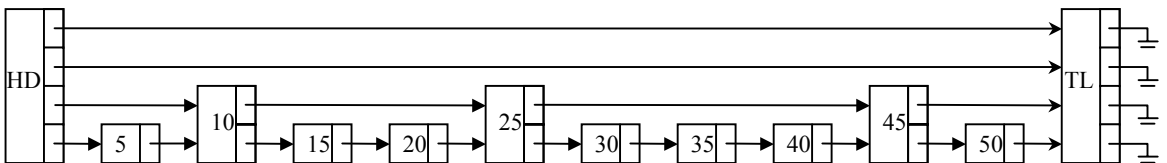



그림 12-10. 1-2-3결정성건너뛰기목록

명백히 아래준위로 떨구기전에 련결상수가 임의의 준위를 따라 순회하게 된다. 결과적으로 탐색시간은 최악의 경우 $O(\log N)$이다.

삽입을 진행하려면 반드시 높이가 H인 새로운 매듭이 삽입될 때 높이가 H이고 매듭이 4개인 간격이 생기지 않는가를 확인해야 한다. 이것은 간단한데 흑적나무에서 적용한것과 류사한 내리전략을 적용한다.

현재 L준위우에서 한개 준위를 떨구려고 한다고 가정하자. 만일 크기 3을 가진 간격에 떨어 지려고 한다면 높이 L을 가지도록 간격에서 중간항목들을 올린다. 그리하여 크기가 1인 2개의 간격이 형성된다. 이것은 삽입앞로정우에서 크기가 3인 간격들을 제거함으로써 중간항목의 높이를 임의로 증가시키는 경우와 같이 삽입이 안전하다는것을 알수 있다.

실례에서 보는것처럼 그림 12-11은 그림 12-10의 결정성건너뛰기목록에 항목 27을 삽입하는것을 보여 준다. 머리부매듭에서 준위 3으로부터 준위 2로 떨구려고 한다. 떨구기는 3간격안에 있게 되므로 중간항목(25)은 높이 3으로 올리고 분리된다. 준위 2에서의 탐색은 준위 1로 떨구려고 하는 25를 취하게 한다. 다시 간격 3이 보이며 35항목은 높이 2로 증가한다. 결과는 그림 12-12에 보여 준다. 27을 삽입하려고 할 때 그것은 그림 12-13에서 보여 주는것처럼 목록에 이어 놓는다.

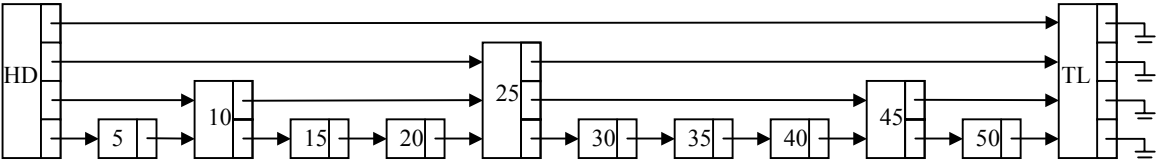



그림 12-11. 매듭 27의 삽입 1: 매듭 25를 들어 올려
높이가 2인 세개의 매듭을 분리

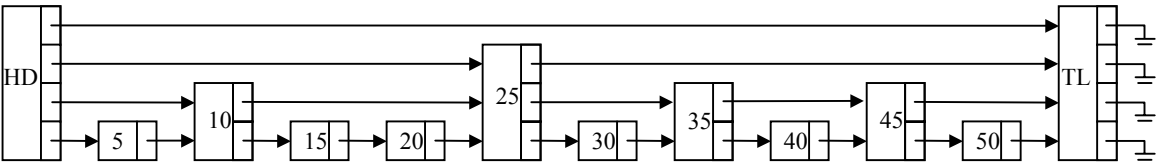



그림 12-12. 매듭 27의 삽입 2: 35를 들어 올려
높이가 1인 세개의 매듭을 분리

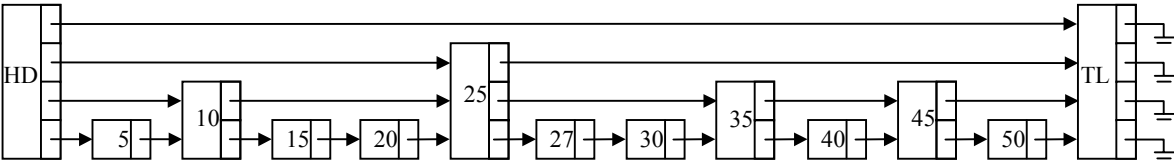



그림 12-13. 매듭 27 의 삽입 3: 27 은 높이가 1 인 매듭으로 삽입된다.

크기가 1인 간격을 가질 때 목록의 제거는 어렵다. 간격이 1인것으로 떨구려고 할 때 린접하고 있는 항목에서 빌리든지(만약 그것이 간격이 1이 아니라면)아니면 린접항목에서 간격을 분리하는 매듭의 높이를 낮추든지 하여 이것을 확장한다. 이 두가지는 간격

이 1인것으로 하여 결과는 크기가 3인 간격으로 된다. 코드는 설명보다 더 복잡하다. 왜냐하면 고려해야 여러가지 경우가 있기때문이다.

그림 12-14. 그림 12-13에서 1-2-3결정성건너뛰기목록의 련결목록실현

이 모든것이 어떻게 실행되겠는가? 모든 설명을 한다음 코드량은 작아 지게 된다는 것을 알게 될것이다.

첫번째로 중요한것은 매듭의 높이 h를 h+1의 높이로 올릴 때 h의 련결을 새로운 배렬로 복사하는데 걸리는 $O(h)$시간을 소비할수 없다는것이다. 다른 한편 삽입하는데 걸리는 림계시간은 $O(\log^2 N)$이다. 합리적인 방법은 련결된 목록에 의하여 높이가 h인 매듭에서 h앞방향련결을 표시하는것이다. 준위 아래로 내려 가기때문에 매듭의 련결된 목록은 준위 h에서부터 앞방향련결로부터 시작하며 준위 1앞에서 앞방향련결을 끌낸다.

두번째로 최적화는 보다 요령적인 문제인데 기억공간의 소비이다. 한개의 항목과 앞방향으로 련결된 목록으로된 항목을 보관시킬대신 앞방향련결목록과 앞방향항목쌍을 기억시킨다. 이것을 실현한것이 그림 12-14이다. 이것이 무엇을 의미하는가를 보여 준 그림 12-14는 그림 12-13을 달리 표현한것이다. 우리는 그림 12-13을 설명하기 위하여 술어 abstract 혹은 logical을 리용하며 또한 실행하려고 할 때에는 그림 12-14를 참고한다.

첫째로 주의할것은 지평선(즉 왼쪽에서 오른쪽으로 훑어 가는 높이들)은 리론적인 표현과 실제적인 처리의 꼬리매듭을 삭제하는것을 제외하고는 같다는것이다. 실행에서 매개 매듭들은 준위를 낮추게 하는 련결, 같은 준위에서 다음 매듭에로의 련결 그리고 론리적으로 다음 항목(개념을 서술한것처럼)에 기억된 항목과의 련결을 보존한다.

어떤 항목들은 한번이상 출현한다. 실례로 25는 세 곳에 출현하였다. 사실 하나의 매듭이 리론적인 고찰에서 높이 h를 가지면 이 항목은 실제적인 실행알고리듬에서 h개의 위치에 출현한다. 중요한 결과와 결론은 그에 대한 처리를 진행한후에 설명하려고 한다.

기본매듭은 하나의 항목과 두개의 련결로 되여 있다. 코드를 더 빨리 더 간단히 작성하기 위하여 꼬리매듭을 리용한다. 즉 이것을 ∞로 주는것이 불가능하거나 원하지 않

는다면 일부 다른 수법을 리용해야 한다. 또한 머리부감시매듭과 꼬리감시매듭이 있어야 하는데 이것은 NULL련결과 치환하기 위한것이다. SkipNode클라스와 DSL자료성원들은 프로그람 12-9에 보여 주었다.

```
template <c1ass Comparab1e>
class SkipNode
{
      Comparable element;
      SkipNode *right;
      SkipNode *down;

      SkipNode( const Comparable & theElement = Comparab1e(    ),
                                   SkipNode *rt= NULL, SkipNode *dt = NULL )

:e1ement( theElement ), right( rt ), down( dt ) { }
      friend class DSL<Comparab1e>;
};

template <c1ass Comparab1e>
class DSL
{
    public:
        explicit DSL( const Comparable & inf );
                // Additional public member functions (not shown)
    private:
                // Data members
        const Comparable INFINITY;
        SkipNode<Comparab1e>    *header; // The list
        SkipNode<Comparab1e>    *bottom;
        SkipNode<Comparab1e>    *tai1;
              // Additional private member functions (not shown)
};
```

프로그람 12-9. 결정성건너뛰기목록: SkipNode 클라스와 DSL 자료성원

탐색기능은 란수화된 건너뛰기목록에서와 같다. 프로그람 12-10은 비교결과가 일치하지 않으면 다음 준위로 내려 오든지 아니면 오른쪽으로 가는데 이것은 비교결과에 관계된다. 프로그람 12-11에 보여 준 삽입은 감시매듭에 의하여 대단히 간단해 진다. 일부 적합지 못한 련결에서 볼수 있는것처럼 매개 련결에 대하여 NULL검사를 진행한다면 코드의 크기는 잠간 3배나 될것이다.

프로그람 12-9에서 지적하는것은 결정성건너뛰기목록의 삽입을 위한 코드는 흑적나무때보다 더 적은 경우를 가지고 더 짧게 작성된다. 최악의 경우에는 두개의 련결과 하나의 항목을 포함하는 2*N*개의 매듭들을 가진다. 흑적나무에서는 두개의 련결과 하나의

항목, 하나의 색비트를 포함하는 *N*개의 매듭들을 가진다. 그래서 2배나 많은 기억공간을 리용할수 있다. 이것은 물론 명백한 결과이다. 첫째로, 실험은 결정성건너뛰기목록은 평균 약 1.57개정도의 매듭들을 가진다는것을 알수 있다. 둘째로, 어떤 경우 결정성건너뛰기목록은 실지로 흑적나무보다 기억공간을 더 적게 쓴다는것을 알수 있다.

```
/**
 *   Find item x in the tree.
 *   Return the matching item, or INFINITY if not found.
 */
template <c1ass Comparab1e>
const Comparable & DSL<Comparab1e>::find( const Comparable & x ) const
{
    SkipNode<Comparab1e> *current = header;
    bottom->e1ement = x;
    for(   ;   ; )
          if( x < current->e1ement )
                current = current->down;
          else if( current->e1ement < x )
                current = current->right;
          else
                return elementAt( current );
}

/**
 * Internal method to get element data member from node t.
 * Return the element data member, or INFINITY if t is at the bottom.
 */
template <c1ass Comparab1e>
const Comparable & DSL<Comparab1e>::
elementAt( SkipNode<Comparab1e> *t ) const
{
          return t = = bottom ? INFINITY : t->e1ement;
}
```

프로그램 12-10. 결정성건너뛰기목록: find루틴

여기에 C나 C++에서 적용한 산 실례가 있다. 32bit콤퓨터에서 지적자와 옹근수형은 4byte이다. UNIX변종을 포함하는 일부 체계에서 기억기는 2의 제곱인 토막으로 배치되며 그러나 2의 토막인 4byte는 기억기관리루틴에 의하여 리용된다. 그러므로 12byte에 요구되는 자료는 16byte기억토막에 채워 지며 따라서 리용자를 위하여 12byte가 배당되고 4byte는 비여 있다. 13byte를 위하여 필요되는것 역시 32byte기억구역이 할당되여야 한다. 그래서 이 경우 결정성건너뛰기목록은 매듭당 16byte를 리용하며 평균 1.57*N*개의 매듭이 있는데 이것들은 25*N*byte를 요구한다. 흑적나무는 32*N*byte수가 리용된다. 이것

은 일부 콤퓨터들에서 허실되는 공간이 대단히 크다는것을 보여 준다. 이것은 자체조직구조(Self-organizing Structures)에 기인된다.

```
/**
 * Insert item x into the DSL.
 */
template <c1ass Comparab1e>
void DSL<Comparab1e>::insert( const Comparable & x )
{
    SkipNode<Comparab1e> *current = header;

    bottom->e1ement = x;
    while ( current != bottom )
    {
        whi1e( current->e1ement < x )
                current = current->right;

        // If gap size is 3 or at bottom level and
        // must insert, then promote middle element
        if( current->down->right->right->e1ement < current->e1ement )
        {
            current->right = new SkipNode<Comparab1e>( current->e1ement,
                        current->right, current->down->right->right );
            current->e1ement = current->down->right->element;
        }
        else
            current = current->down;
      }

      // Raise height of DSL if necessary
      if( header->right != tail )
        header = new SkipNode<Comparab1e>( INFINITY, tail, header );
}
```

프로그람 12-11. 결정성건너뛰기목록: 삽입루틴

결정성건너뛰기목록의 수행은 흑적나무와 비교하기에 아주 유리하다. 삽입할 때에 개선되였다고 보아 지는 코드행은 아래와 같다.

if(current->down->right->right->element < current->element)

가령 우의 배렬에서 세개의 요소에 항목들을 기억시키고 이 세번째 항목에 직접 접근할수 있다면 두개의 right지적자를 리용하는것보다 더 낫다. 그림 12-15는 결과적인 구조를 보여 주는데 이것은 반대로 제4장에서 서술한 B-나무와 아주 류사한 느낌을 준다. 이것을 1-2-3결정성건너뛰기목록의 수평배렬실현(horizontal array implementation)이라고 한다.

B-나무에 고차B-나무가 있는것처럼 두개의 련결된 목록과 수평배렬형태들에서 고차결정성건너뛰기목록을 가질수 있다. 이 방법들은 아직 더 연구되여야 하며 전문체계와 응용프로그람에 완전히 의거하게 된다.

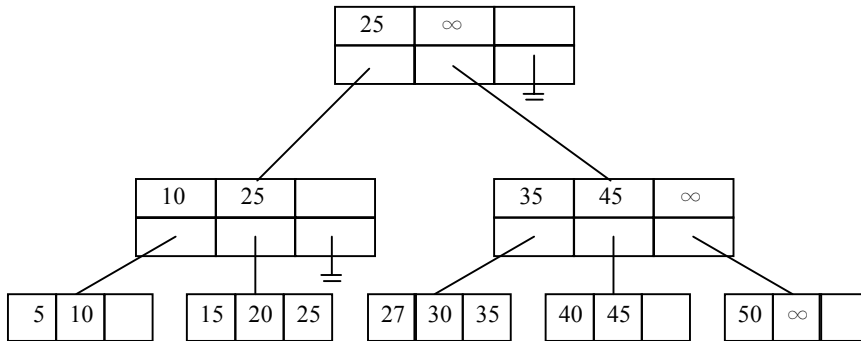



그림 12-15. 그림 12-14의 수평배렬실현

제4절. AA-나무

가능한 회전수가 많은것으로 하여 흑적나무는 코드작성에서 대단히 까다로으며 특히 삭제는 더하다. 결정성건너뛰기목록은 코드량이 더 적지만 요구되는 세개의 표식에 의하여 지적되는것만큼 이것 역시 까다롭다. 결정성건너뛰기목록에서 삭제는 확실히 간단치 않은 과제이다. 이 절에서 **BB나무**(*BB tree*)라고 하는 **2진B나무**(*bynary B-tree*)의 간단하고 실제적인 실현에 대하여 본다. **BB**나무는 한개의 보충적인 조건이 있는 흑적나무이다. 즉 매듭은 많아서 한개의 붉은색자식을 가질수 있다. 코드작성을 쉽게 하기 위하여 다음의 규칙을 작성한다.

① 먼저 오른쪽 자식만이 붉은색이 될수 있다는 조건을 첨부한다. 이것은 있을수 있는 재구축과정의 거의 절반을 축감시킨다. 이것은 또한 삭제알고리듬에서 애먹던 과정을 축소시키는것으로 된다. 가령 내부매듭이 하나의 자식매듭만을 가지면 그 자식매듭은 오른쪽 자식(붉은색매듭으로 나타나는)이여야 한다. 왜냐하면 검은색의 왼쪽 자식은 흑적나무의 4번째 조건에 어긋나기때문이다. 그래서 이 오른쪽 부분나무에서의 최소매듭으로 내부매듭을 치환할수 있다.

② 절차(방법)들은 재귀적이다.

③ 매개 매듭의 색비트를 기억시킬대신 가장 작은 옹근수(실례로 8bit)로 정보를 기억시킨다. 이 정보는 매듭의 준위이다.

매듭의 준위(level)는

- 매듭이 잎이면 1
- 매듭이 붉은색이면 그 준위는 부모의 준위와 같다.
- 매듭이 흑색이면 그의 부모의 준위보다 1적다.

이 결과는 하나의 AA나무이다. 프로그람 12-12는 AA나무에서 리용되는 형정의를 보여 준다. 다시 한번 표식을 리용하여 NULL을 표시한다.

```
template <c1ass Comparab1e>
class AANode
{
      Comparable element;
      AANode          *left;
      AANode         * right;
      int                   level;

      AANode(    ) : 1eft( NULL ),    right( NULL ),    level( 1 ) { }
      AANode( const Comparable & e,    AANode    *lt,    AANode    * rt,    int 1v = 1 )
         : e1ement( e ),    1eft( 1t ),    right( rt ),    1eve1( 1v ) { }

      friend class AATree<Comparab1e>;
};

/**
  * Construct the tree.
                            */
template <c1ass Comparab1e>
AATree<Comparab1e>::AATree( const Comparable & notFound )
      : ITEM_NOT,FOUND( notFound )
   {
         nullNode = new AANode<Comparab1e>;
         nu11Node->1eft = nullNode->right = nullNode;
         nu11Node->1eve1 = 0;
         root = nullNode;
   }
```

프로그람 12-12. AA-나무들. 매듭클라스와 AA나무의 초기화

만일 AA구조를 색갈로부터 준위로 변환한다면 명백히 왼쪽 자식은 그의 부모매듭보다 한준위 더 낮으며 오른쪽 자식은 0이거나 그의 부모보다 한준위 더 낮다(그러나 더 크지 않다.).

수평병합은 매듭과 같은 준위의 자식매듭들사이의 련결이다. 즉 구조는 수평련결이 오른쪽 련결이며 여기에 두개의 련속적인 수평련결이 되지 말아야 한다는 요구를 제기한다. 그림 12-16은 간단한 AA나무를 보여 준다. 탐색은 보통의 알고리듬을 써서 진행한다. 새로운 항목의 삽입은 언제나 맨 밑준위에서 진행한다. 그러나 두가지 문제점이 생길수

있는데 2의 삽입은 왼쪽 수평련결을 발생시키며 한편 45의 삽입은 두개의 련속적인 오른쪽 련결을 발생한다는것이다.

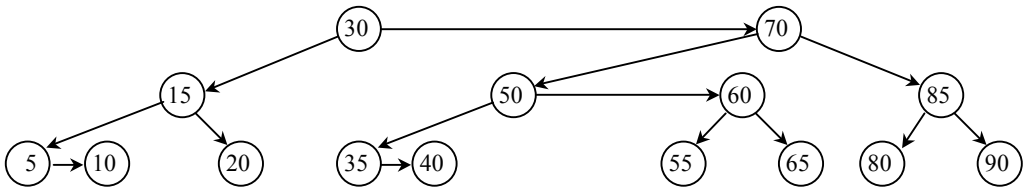



그림 12-16. 매듭을 삽입한 AA 나무결과

두 경우에 다 간단히 회전만을 진행하여 문제를 해결할수 있다. 즉 오른쪽회전으로 왼쪽 수평련결을 없애며 왼쪽회전으로 련속적인 오른쪽 수평련결을 없앤다. 이 과정을 각각 Skew와 분리(Split)라고 한다. 프로그람 12-13은 이 과정에 대한 코드를 보여 주었다. Skew는 왼쪽 수평련결을 없애지만 오른쪽의 련속적인 련결들을 만들어 낼수 있다. 그러므로 먼저 Skew를 진행하고 다음 분리를 진행한다. 분리한후 중간매듭 *R*는 준위가 증가한다. 이것은 왼쪽 수평매듭 혹은 련속적인 오른쪽 매듭들을 만들어 냄으로써 원래의 *X*의 부모매듭에 대한 문제를 발생시킬수 있다. 두 경우의 문제점들은 Skew/Split전략을 써서 풀수 있다. 이것은 재귀를 쓰면 자동적으로 진행된다. 그림 12-17은 두개의 방법을 다 서술한다.

```
/**
 * Skew primitive for AA-trees.
 * t is the node that roots the tree.
 */
template <c1ass Comparab1e>
void AATree<Comparab1e>::skew( AANode<Comparab1e> * & t ) const
{
      if( t->left->1eve1 == t->level )
            rotateWithLeftChild( t );
}

/**
* Split primitive for AA-trees.
* t is the node that roots the tree.
*/

template <c1ass Comparab1e>
void AATree<Comparab1e>: :sp1it( AANode<Comparab1e> * & t) const
{
       if( t->right->right->1eve1 == t->1eve1 )
       {
```

```
            rotateWithRightChild( t );
            t->level++;
        }
    }
```

프로그람 12-13. AA-나무들 Skew와 Split방법들

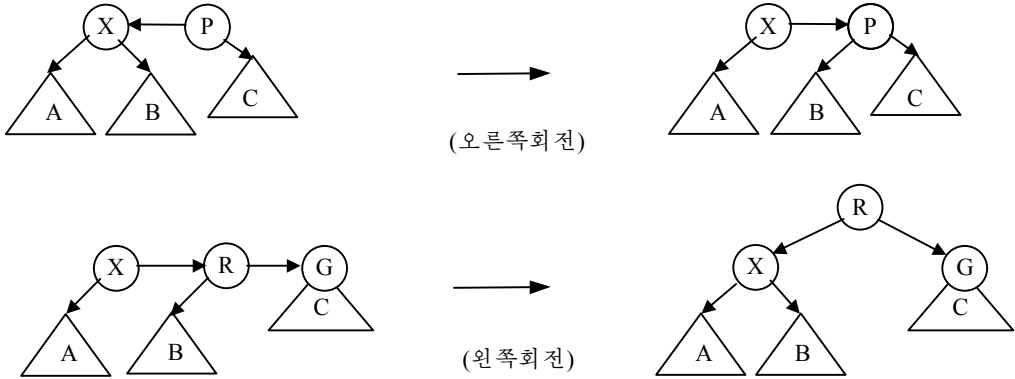



그림 12-17. Skew 와 Slpit, R 의 준위는 분리하면 증가한다.

그림 12-16에서 AA나무에 45를 삽입할 때 진행되는 동작을 그림 12-18~12-22에서 보여 주었다.

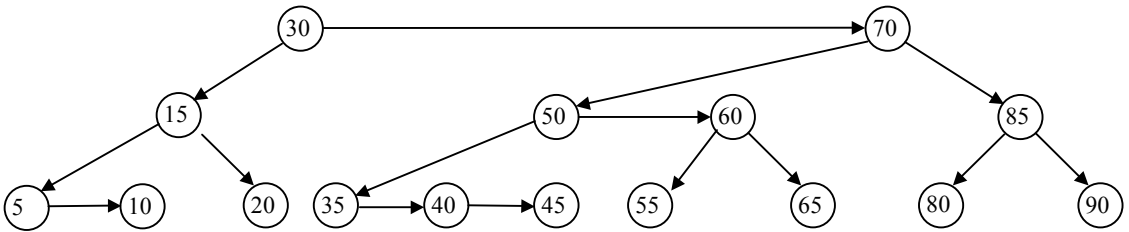



그림 12-18. 간단한 나무에 45를 삽입한후

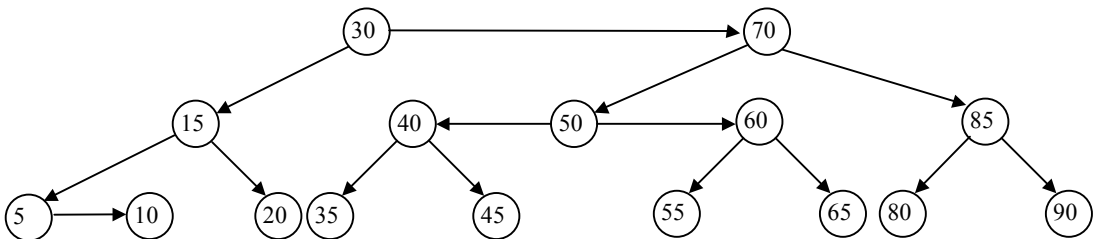



그림 12-19. 35에서 분리된후

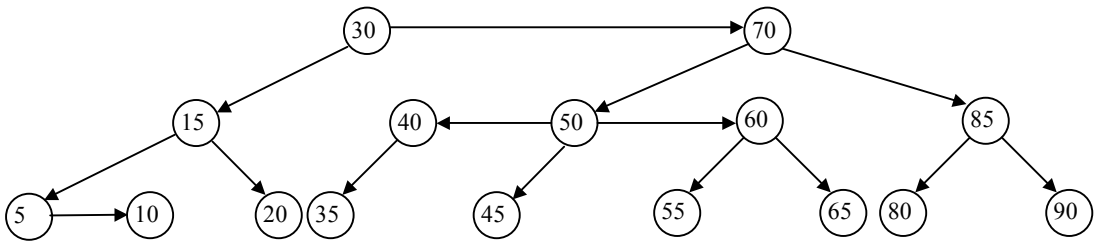



그림 12-20. 50에서 경사진후

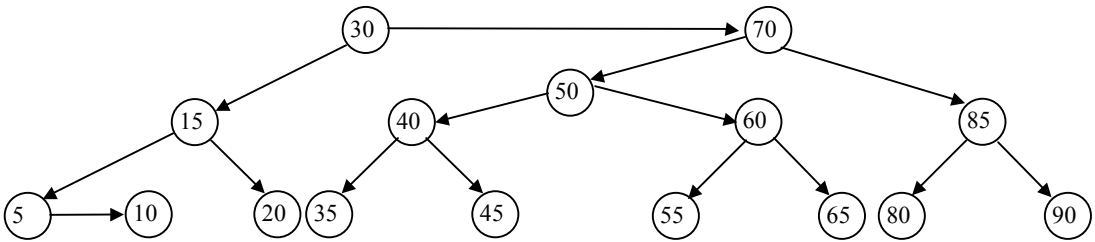



그림 12-21. 40에서 분리된후

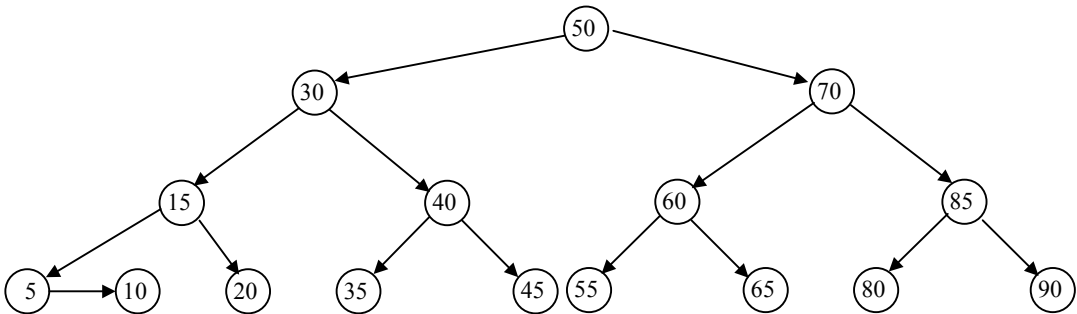



그림 12-22. 70에서 skew하고 30에서 분리된 결과나무

프로그람 12-14에서 보여 준바와 같이 삽입은 비평형구조의 실현때보다 2행 더 많을뿐이다. 삭제는 물론 보다 복잡하지만 많은 특수한 경우들이 제거되였으므로 코드는 실제로 간단히 작성할수 있다. 무엇보다먼저 매듭이 잎이 아니면 오른쪽 자식을 가져야 한다는것을 상기시킨다. 이것은 매듭을 삭제할 때 오른쪽 부분나무에서 매듭을 가장 작은 자식으로 치환할수 있다는것을 의미하는데 이 부분나무는 준위 1에 위치하고 있다는것이 담보된다. 방조를 위하여 두개의 클라스변수 deletedNode와 lastNode를 리용한다. 이것들은 remove연산이 재귀산법이기때문에 정적이여야 한다. 오른쪽 련결을 순회할 때 deleteNode방법을 조종한다. 왜냐하면 나무의 밑에 도착할 때까지 remove방법을 재귀적으

로 호출하여 제거되는 나무가 있으면 deleteNode가 그것을 포함하는 매듭을 가리키도록 하기 위해서이다. 항목이 나무에 있으면 나무의 맨 밑에 도착할 때까지 정지되지 않기때문에 lastNode는 탐색이 끝나게 되는 잎을 가리킨다. 나무의 밑에 도달할 때까지 정지하지 않으므로 항목이 나무에 있다면 lastNode는 치환되는 값을 포함하는 준위 1의 매듭을 가리키게 되며 나무에서 제거되여야 한다.

```
/**
 * Internal method to insert into a subtree.
 * x is the item to insert.
 * t 1s the node that roots the tree.
 * Set the new root.
 */
template <c1ass Comparab1e>
void AATree<Comparab1e>::
Insert( const Comparable &    x, AANode<Comparab1e> *    & t );
{
      if( t == nullNode )
          t = new AANode<Comparab1e>( x, nullNode, nu11Node );
      else if( x < t->e1ement )
          insert( x, t->1eft );
      else if( t->e1ement < x )
          insert( x, t->right );
      else
          return; // Duplicate; do nothing
      skew( t );
      split( t );
}
```

프로그람 12-14. AA-나무들: 삽입방법

나무의 맨 밑에 이르면 단계 2를 수행하는데 이것은 준위 1의 매듭값을 내부매듭에 복사하고 그다음은 준위 1의 매듭을 무시한다. 잎이 아닌 매듭들은 재귀적인 호출에 의하여 그것들의 준위가 파괴되였는가를 검사한다. *T*가 현재 매듭이라고 하자. 만일 삭제가 *T*의 자식매듭들중의 하나의 준위(재귀순환에 들어 간 자식매듭만이 실제로 영향을 받게 되지만 간단히 하기 위하여 그것을 따지지 않는다.)를 T준위보다 2만큼 낮춘다면 *T*의 준위역시 낮추어야 한다. 더 나아가서 *T*준위에 오른쪽 붉은자식이 있으면 *T*의 오른쪽 자식은 낮아 진 준위를 가져야 한다. 이 점에서 같은 준위상에 6개의 매듭들을 가질수 있다. 즉 *T*와 *T*의 오른쪽 붉은자식 *R*, *R*의 두 자식들과 그 자식들의 오른쪽 붉은자식들이다. 그림 12-23에 가장 간단한 방법을 보여 주었다.

매듭 1이 삭제된후 매듭 2와 5는 준위1의 매듭들로 된다. 먼저 매듭 5와 3사이에 지금 련결된 왼쪽 수평련결을 수정하여야 한다. 이것은 본질적으로 2번의 회전(하나는 5와

3사이, 다른 하나는 5와 4사이)을 요구한다. 이 경우 현재매듭 *T*는 포함되지 않는다. 다른 한편 삭제가 오른쪽에서부터 시작되면 *T*의 왼쪽 매듭은 즉시 수평련결로 된다. 이것역시 류사한 2중회전(*T*에서 시작하는)을 요구하게 된다.

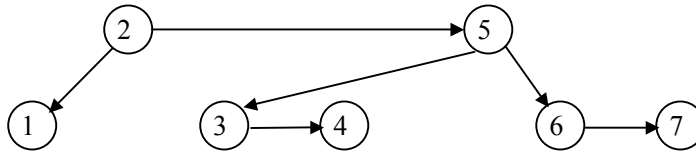



그림 12-23. 1이 제거될 때 모든 매듭들은 수평잎련결을 유도하는 준위 1로 된다. 오른쪽에로의 련결은 Skew를 3번 호출하여 실현하며 split를 2번 호출하여 수평련결을 제거

이 모든 경우를 시험해 보는것을 피하기 위하여 Skew를 3번 호출한다. 일단 이렇게 하면 Split에 대한 두번의 호출은 수평의 끌들을 재배치하는데 충분하다. 이 전체적인 삭제루틴을 프로그람 12-15에서 보여 준다. 어쨌든 최종적으로 이것은 상대적으로 코드작성이 쉬운 자료구조이다.

```
/**
  * Internal method to remove from a subtree.
  * x    is the item to remove.
  * t is the node that roots the tree.
  * Set the new root.
  */
template <c1ass     Comparab1e>
void AATree<Comparab1e>::
remove( const Comparable & x, AANode<Comparab1e> * & t )
{
      static AANode<Comparab1e> *lastNode,    *deletedNode = nullNode;
      if( t ! = nullNode )
      {
         // Step 1: Search down the tree and set lastNode and deletedNode
lastNode = t;
        if( x < t->e1ement )
              remove( x, t->1eft );
        else
        {
              deletedNode = t;
              remove( x, t->right );
                 }
        // Step 2: If at the bottom of the tree and
        //            x is present, we remove    it
                 if( t = = testNode )
```

```
    {
          if( deletedNode == nullNode || x != de1etedNode->element )
                return;         // Item not found; do nothing
          deletedNode->element = t->element;
          deletedNode = nullNode;
          t = t->right;
          delete lastNode;
    }
    // Step 3: Otherwise, we are not at the bottom; rebalance
    else
          if( t->1eft->1eve1 < t->1eve1 - 1 ||
              t->right->1eve1 < t->1eve1 - 1 )
          {
                if( t->right->1eve1 > t->1eve1 )
                    t->right->1eve1 = t->1eve1;
                skew( t );
                skew( t->right );
                skew( t->right->right );
                sp1it( t );
                sp1it( t->right );
          }
    }
}
```

프로그람 12-15. AA-나무들: 제거과정

제5절. 트리프

2진탐색나무의 마지막형태 **트리프**는 아마 모든 자료구조가운데서 제일 간단할것이다. 건너뛰기목록처럼 란수를 리용하며 임의의 입구에 대하여 $O(\log N)$시간특성을 준다. 탐색시간은 비평형2진탐색나무(평형탐색나무보다 느리다.)와 같으며 삽입시간은 재귀적인 비평형2진탐색나무의 실행시간보다는 약간 느리다. 비록 삭제가 더 느리다 할지라도 이것역시 기대시간은 $O(\log N)$이다.

트리프는 그림없이도 설명할수 있을만큼 간단하다. 나무에서 매개 매듭은 하나의 항목과 오른쪽과 왼쪽 지적자들, 매듭이 생성될 때 우연적으로 할당되는 우선권을 가지고 있다. 트리프는 매듭의 우선권이 더미순서를 만족시키는 즉 매듭의 우선권이 적어도 그의 부모매듭의 우선권만큼 큰 특성을 가지는 2진탐색나무이다.

매개가 구별되는 우선권을 가지는 항목들의 모임을 한개만의 트리프로써 표시할수 있다. 이것은 유도방법으로 쉽게 증명할수 있다. 왜냐하면 제일 낮은 우선권을 가진 매듭이 뿌리로 되여야 하기때문이다. 결과 나무는 $N!$항목의 순서화대신에 $N!$개의 가능한 우선권의 배렬들을 기초로 형성된다.

매듭은 직접 선언하는데 다만 priority자료성원을 추가할것을 요구한다. 프로그람 12-16에서 보여 준바와 같이 표식 nullNode는 ∞의 우선권을 가지게 된다. 트리프에로의 삽입은 간단하다. 즉 항목이 잎으로 추가된후에 트리프가 그의 우선권이 더미순서를 만족시킬 때까지 트리프에로 그것을 회전시킨다. 이것은 기대되는 회전수가 2보다 작다는 것을 보여 준다. 삭제할 항목이 발견된 다음에는 그의 우선권을 ∞로 증가시키고 낮은 우선권을 가진 자식쪽의 경로를 따라 그 항목을 회전시켜 삭제할수 있다. 일단 그것이 잎이 되면 제거될수 있다. 프로그람 12-17과 12-18에서 실행하는 루틴은 재귀를 리용하여 이 전략을 실현한다. 여기에서 비재귀적인 실현은 독자들에게 맡긴다(련습문제 12-17) 삭제에 대하여 강조할것은 매듭이 론리적으로 잎일 때에도 그것은 여전히 두개의 왼쪽, 오른쪽 자식으로서 nullNode를 가진다는것이다. 결과 이것은 오른쪽 자식과 함께 회전한다. 회전한 다음 t는 nullNode이고 왼쪽 자식 즉 이제 삭제하는 항목을 기억하고 있는 자식은 해방된다. 여기에서 언급할것은 반복이 없다는 가정하에서 즉 그렇게 되지 않으면 remove가 실패할수 있다(왜 그런가?)는것이다.

```
template <c1ass Comparab1e>
class Treap
{
    public:
        explicit Treap ( const Comparable & notFound );
        Treap ( const Treap & rhs );
        ~Treap ( );
            // Additional public member functions (not shown)
    private:
      Treap Node<Comparab1e>    *root;
        const Comparable ITEM_NOT_FOUND;
        Treap Node<Comparab1e>    *nullNode;
        Random randomNums;
            // Additional private member functions (not shown)
};
/**
* Construct the Treap.
*/
template <c1ass Comparab1e>
Treap <Comparab1e>:: Treap ( const Comparable & notFound )
      : ITEM_NOT_FOUND( notFound )
{
      null Node = new Treap Node<Comparab1e>;
      nullNode->1eft = nu11Node->right = nullNode;
      nullNode->priority = INT_MAX;
      root = nu1lNode;
}
```

프로그람 12-16. 트리프클라스의 대면부와 구축자

```
/**
* Internal method to insert into a subtree.
* x is the item to insert.
* t is the node that roots the tree.
* Set the new root.
* (randomNums is a Random object that is a data member of Treap.)
*/
template <c1ass Comparab1e>
void    Treap <Comparab1e>::
insert( const Comparable & x, Treap Node<Comparab1e> * & t )
{
     if( t = = nu11Node )
     t = new Treap Node<Comparab1e>( x, nullNode, nu11Node,
                                                  randomNums.randomInt( ) );
}
else if( x < t->e1ement )
{
       insert( x, t->1eft );
        if( t->1eft->priority < t->priority )
                rotateWithLeftChild( t );
 }
 else if( t->e1ement < x )
 {
        insert( x, t->right );
        if( t->right-> priority    < t->priority )
                 rotatewithRightChi1d( t );
   }
   // else duplicate; do nothing
}
```

프로그램 12-17. Treps삽입루틴

```
/**
  * Internal method to remove from a subtree.
  * x is the item to remove.
  * t is the node that roots the tree.
  * Set the new root.
  */
template <c1ass Comparab1e>
void Treap <Comparab1e>::remove( const Comparable & x,
                               Treap Node<Comparab1e> * & t )
{
      if( t != nullNode )
{
          if( x < t->e1ement )
                remove( x, t->1eft );
else if( t->e1ement < x )
```

```
            remove( x, t->right );
        else
        {
            // Match found
        if( t->left->priority < t->right->priority )
                rotateWithLeftChi1d( t );
        else
                rotateWithRightChild( t );

        if( t != nullNode )                    // Continue on down
            remove( x, t );
        else
        {
            delete t->1eft;                // Free the matched node
            t->left = nullNode;            // Fix nullNode
         }
      }
    }
  }
```

프로그람 12-18. 트리프s: 삭제공정

트리프는 priority자료성원을 조정하지 않아도 되므로 실현하기가 매우 쉽다. 평형나무에 접근하는데서 어려운것의 하나는 연산과정에 평형정보를 갱신하는것이 실패로부터 생기는 오유를 추적해 내려 가기가 힘들다는것이다. 트리프는 전체 행의 항목들에서 있을수 있는 삽입과 삭제묶음에 대하여 특별히 비재귀적으로 실현함으로써 비교적 괜찮은 방법이라고 볼수 있다.

제6절. *K*차원나무

어떤 광고회사가 자료기지를 정리하고 일부 주민들에 대한 우편표식을 하려고 한다고 하자. 일반적인 요구는 우편을 34～49살사이에 있으면서 수입이 100000$~150000$사이에 있는 사람들에게 보내려고 하는것이다. 이 문제를 2차원범위질문이라고 한다. 1차원에서 이 문제는 간단히 재귀적인 알고리듬을 써서 $O(M+\log N)$평균시간에 실행하며 미리 구축된 2진탐색나무를 순회하여 풀수 있다. 2차원 혹은 다차원에 대하여 류사한 경계점을 얻을수 있다. 여기서 M은 질문에 대하여 통지된 정합수이다.

2차원 혹은 다차원에 대하여도 류사한 경계점을 얻으려고 한다. 2차원탐색나무는 간단한 특성 즉 홀수준위에서 가지자르기는 첫번째 열쇠를 고려하여 진행하며, 짝수준위에서의 가지자르기는 두번째 열쇠를 고려하여 진행하는 특성을 가지고 있다. 뿌리는 홀수

준위로 되도록 임의로 선정한다. 그림 12-24는 2차원나무를 보여 준다. 2차원나무에로의 삽입은 2진탐색나무에로의 삽입을 단순히 확장한것이다. 즉 나무아래로 내려 갈 때 현재 준위를 보존해야 한다. 코드를 간단히 작성하기 위하여 기준항목은 두개의 요소들의 배렬이라고 하자. 다음 준위는 0과 1사이에 있도록 한다. 프로그람 12-19는 삽입하는 과정을 코드로 보여 준다. 이 부분에서는 재귀를 쓴다. 실천에서 쓰이게 되는 비재귀적인 실현은 선형적이며 련습문제 12-33에서 보게 된다. 한가지 어려운 점은 중복인바 특히 여러가지 항목들이 하나의 열쇠값을 가질수 있다. 코드는 중복을 허용하며 그것들은 늘 오른쪽 가지들에 놓인다. 명백히 이것은 너무 많은 중복을 가지게 되면 문제가 있다.

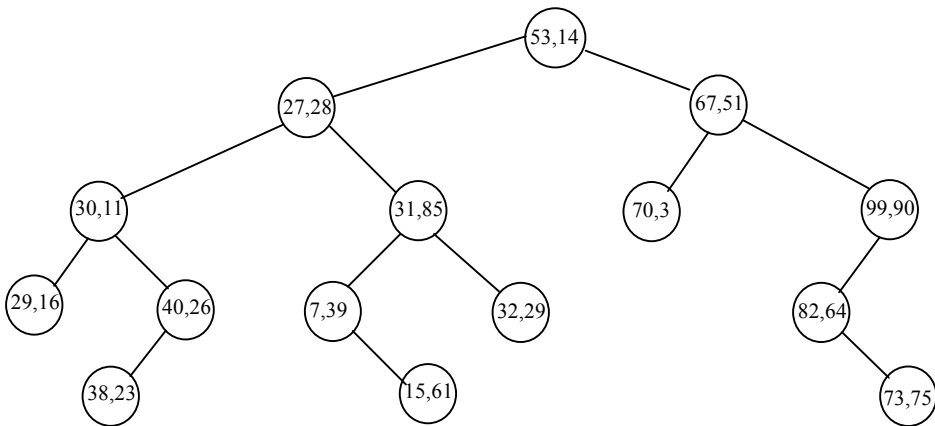



그림 12-24. 간단한 2차원나무

```
template <c1ass Comparab1e>
void KdTree<Comparab1e>::insert( const vector<Comparab1e> & x )
{
    insert( x, root, 0 );
}

template <c1ass Comparab1e>
void KdTree<Comparab1e>::insert( const vector<Comparab1e> & x,
                                 KdNode * & t,    int level )
{
        if( t == NULL )
            t = new KdNode( x );
        else if( x[ level ] < t->data[ level ] )
            insert( x, t->left, 1 - level );
        else
            insert( x, t->right, 1 - level );
}
```

프로그람 12-19. 2차원나무에 삽입

얼핏 생각해 보면 무질서하게 구성된 2차원나무는 무질서한 2진탐색나무와 같은 구조적특성을 가진다는것을 알수 있다. 즉 높이가 평균 $O(\log N)$이고 최악의 경우는 $O(N)$이다.

명백히 $O(\log N)$최악의 경우의 변종들이 존재하는 2진탐색나무와는 달리 균형2차원나무를 담보한다고 하는 도식은 없다. 문제는 그러한 도식은 나무회전에 기초하고 있는데 2차원나무에서 나무회전은 실현할수 없다. 그렇게 할수 있는 가장 좋은 방안은 련습에 설명한것처럼 부분나무를 다시 구성함으로써 나무를 주기적으로 재균형맞추기하는것이다. 명백하게 지연삭제전략을 릉가하는 삭제알고리듬은 없다. 만일 모든 항목들이 질문조작을 조작하기전에 출현하면 완전한 평형2차원나무를 $O(N\log N)$시간동안에 구축할수 있다. 이것을 련습문제 12-21 ㄷ에서 실현해 본다.

2차원나무에서는 여러가지 종류의 질문이 제기된다. 완전한 정합을 요구할수도 있고 두개의 열쇠중 어느 하나에 기준하여 일치하는 기준한 정합을 요구할수 있는데 후자를 부분정합질문이라고 한다. 이것은 둘다 범위질문(직교)의 특수경우들이다.

직교적인 범위질문은 첫번째 열쇠가 지적된 값모임에 있고 두번째 열쇠가 지적된 또 다른 값범위안에 들어 있는 모든 항목들을 준다. 이것은 이 부분의 안내에서 정확히 설명된 문제이다. 프로그람 12-20에서 보여 준것처럼 범위질문은 재귀나무를 순회함으로써 쉽게 해결된다. 재귀적인 호출을 하기전에 검사하여 모든 매듭들을 불필요하게 순회하는것을 피할수 있다.

렬거된 항목을 찾기 위하여 low = high = 찾고 있는 항목으로 설정할수 있다. 부분적으로 일치하는 질문을 수행하기 위하여 열쇠의 범위가 ∞, -∞로 되지 않도록 설정한다. 다른 범위는 낮은 값과 높은 값이 일치되는 열쇠의 값과 같게 되도록 모임을 설정한다.

2차원나무에서 삽입 또는 완전히 일치되는 탐색은 나무의 깊이에 비례되는 시간이 걸린다. 즉 평균은 $O(\log N)$이고 최악의 경우에 $O(N)$이다. 범위탐색의 실행시간은 나무가 얼마나 균형되였는가, 부분일치가 요구되는가, 얼마나 많은 항목들을 실제로 찾으려고 하는가에 관계된다. 방금 보여 준 세가지 결과를 언급한다.

완전히 균형된 나무에 대하여 범위질문은 M개가 일치될 때 최악의 경우에 $O(M+\sqrt{N})$만한 시간이 걸린다. 임의의 매듭에서 다음의 식 $T(N)=2T(N/4)+O(1)$으로 유도되는 4개의 자식매듭중 2개를 보게 된다. 실천에서는 이러한 탐색들이 매우 유효하며 지어 최악의 경우에는 서툴더라도 유효하다. 왜냐하면 일반적인 N에 대하여 $\sqrt{N}$과 logN사이의 차는 Big-oh표에 표기되여 있는 가장 작은 상수에 의하여 보상되기때문이다.

임의로 구성된 나무에 대하여 부분정합질문의 실행시간은 평균 $O(M+N^{\alpha})$이다. 여기서 $\alpha=(-3+\sqrt{17})/2$(다음을 보시오.)이다. 결과는 이것이 기본적인 자유2차원나무의 범위탐색의 평균실행시간을 표시한다는것이다.

***k*차원나무(*k-d Tree*)**에 대하여서도 같은 알고리듬이 적용된다. 즉 매개 준위에서 열쇠들을 통하여 회전하면서 순회한다. 그러나 현실에서 균형은 더 어렵게 얻어 진다. 왜냐면 중복과 비란수적인 입구의 효과가 특징적으로 더 두드러 지기때문이다. 상세한 코드작성은 과제로 맡기고 분석결과를 언급한다. 완전히 균형인 나무에 대하여 범위질문의 최악의 실행시간은 $O(M+kN^{1-1/k})$이다. 임의로 구축된 k차원나무에서 k개중 p개를 포함하는 부분정합질문은 $O(M+N^{\alpha})$시간 걸리는데 여기서 α는

$$(2+\alpha)^p\,(1+\alpha)^{k-p} = 2^k$$

의 정의 뿌리(유일한)이다. 각이한 p와 k에 대한 α의 계산은 련습문제로 남겨 둔다. k=2, p=1일 때의 값이 자유2차원나무에서의 부분정합에 대하여 우에서 설명된 결과로 반영된다.

비록 범위탐색을 반영하는 여러가지 자료구조들이 있겠지만 k차원나무는 기대할만한 실행시간을 얻는 가장 간단한 구조이다.

```
/**
 * Print items satisfying
 * 1ow[ 0 ] <= x[ 0 ] <= high[ 0 ] and
 * 1ow[ 1 ] <= x[ 1 ] <= high[ 1 ]
 */
template <c1ass Comparab1e>
void KdTree<:Comparab1e>::printRange( const vector-<Comparab1e> & low,
                                      const vector<Comparab1e> & high ) const
{
        printRange( tow, high, root, 0 );
}
template <c1ass Comparab1e>
void KdTree<:Comparab1e>::printRange( const vector<Comparab1e> & low,
                                      const vector<Comparab1e> & high,
                                      KdNode *t, int level ) const
{
       if( t != NULL )
       {
              if( 1ow[ 0 ] <= t->data[ 0 ] && high[ 0 ] >= t->data[ 0 ] &&
              low[ 1 ] <= t->data[ 1 ] && high[ 1 ] >= t->data[ 1 ] )
              cout « "(" « t->data[ 0 ] « ","
                         « t->data[ 1 ] « ")" « endl;
              if( 1ow[ level ] <= t->data[ level ] )
                  printRange( low, high, t->1eft, 1 - level );
              if( high[ level ] >= t->data[ level ] )
                  printRange( low, high, t->right, 1 - level );
       }
}
```

프로그람 12-20. 2차원나무: 범위탐색

제7절. 쌍더미

이제 고찰할 마지막자료구조는 **쌍더미***(Pairing heaps)*이다. 이 쌍더미의 분석은 아직 진행중이지만 decreaseKey연산을 필요로 할 때에는 다른 더미구조체들보다 완전히 성능이 높다. 쌍더미가 효률적인것으로 되는 중요한 리유는 그것이 간단하다는것이다. 쌍더미는 더미가 순서화된 나무로 표현된다. 그림 12-25는 간단한 쌍더미를 보여 준다.

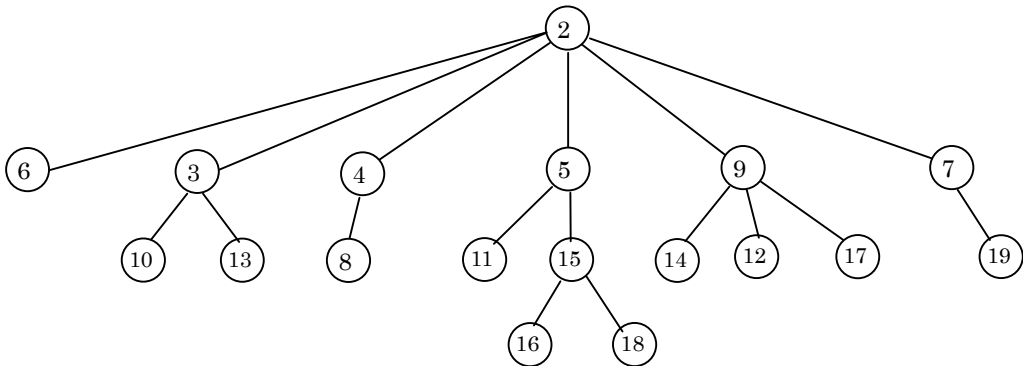



그림 12-25. 간단한 쌍더미

실제적인 쌍더미의 실현은 제4장에서 설명한것처럼 왼쪽 자식, 오른쪽 형제의 표현을 리용한다. 이제 보게 되지만 decreaseKey연산은 매개 매듭이 추가적인 련결을 포함할것을 요구한다. 제일 왼쪽의 자식매듭은 그의 부모에 련결된다. 반대쪽 매듭은 오른쪽 형제이며 그의 왼쪽 형제와 련결된다. 이 자료성원을 prev라고 하자. 간단히 하기 위하여 클라스구조와 쌍더미매듭정의는 잠시 빼버렸다. 그것들은 완전히 선형적이다. 그림 12-26은 그림 12-25에서의 쌍더미의 실제적인 표현을 보여 준다.

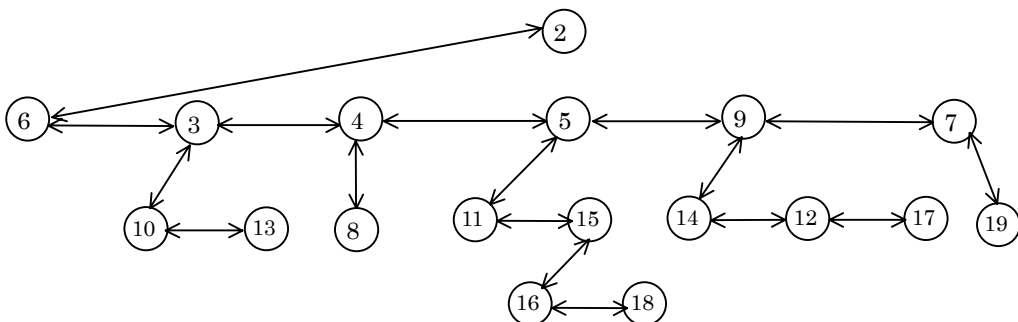



그림 12-26. 앞의 쌍더미의 실제적인 표현

기본조작들은 구도잡는것으로 시작한다. 두개의 쌍더미를 합하기 위해 큰 뿌리를 가지는 더미를 보다 작은 뿌리를 가지는 더미의 왼쪽 자식으로 만든다. 삽입은 보통 합치기의 특이한 경우이다. decreaseKey를 수행하려면 요구되는 매듭에서 값을 낮추어야 한다.

모든 매듭에 대하여 부모지시기를 보존하지 않으므로 이것이 더미순서에 맞는지 안맞는지는 모른다. 따라서 조정되는 매듭을 그의 부모로부터 떼여 내고 그때 생기는 2개의 더미들을 합하는 방법으로 decreaseKey를 완성한다. deleteMin을 수행하기 위하여 더미를 수집해 가면서 뿌리를 제거한다. 뿌리의 자식이 c개 있다면 병합절차를 c−1번 호출하여 더미들을 다시 모을수 있을것이다. 가장 중요한것은 병합을 실현하는데 리용되는 방법과 c−1개의 병합에 그 방법이 어떻게 적용되는가 하는것을 상세히 설명하는것이다.

그림 12-27은 2개의 보조더미가 어떻게 병합되는가를 보여 준다. 이 공정은 두번째 보조더미가 형제들을 가지도록 일반화되여 있다. 앞에서 본것처럼 보다 큰 뿌리를 가진 보조더미는 다른 보조더미의 제일 왼쪽 자식으로 만들어 진다. 코드는 프로그람 12-21에서 보여 준다. 지적자가 그의 prev자료성원을 할당하기전에 NULL에 대하여 검사를 받는 여러번의 기회가 있다는것을 이야기한다. 이것은 이 장의 탐색나무실현에서 늘 씌였던 NullNode표식을 가지고 있는것이 쓸모가 있다는것을 암시한다.

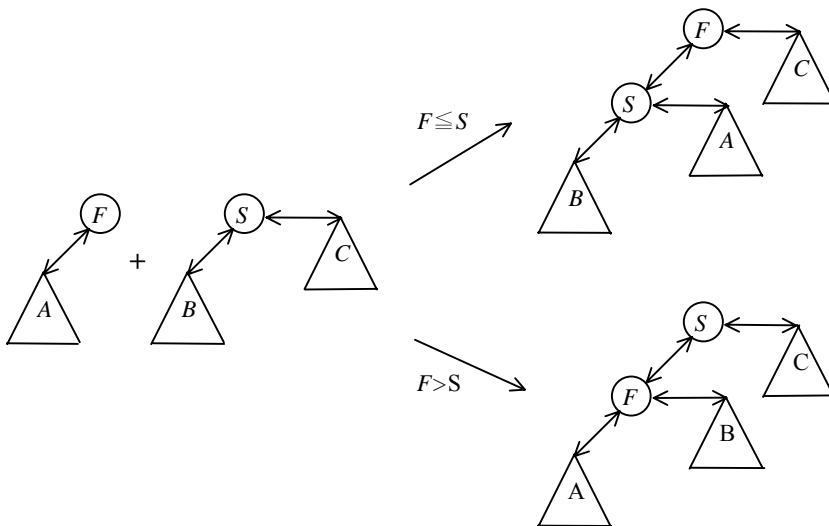



그림 12-27. 두개의 보조더미의 CompareAndLink 련결

```
/**
 * Internal method that 1s the basic operation to maintain order,
 * Links first and second together to satisfy heap order.
 * first is root of tree 1, which may not be NULL.
 * first->nextSib1ing MUST be NULL on entry.
* second is root of tree 2, which may be NULL.
```

```
 * first becomes the result of the tree merge.
 */
template <c1ass Comparab1e>
void   PairingHeap<Comparab1e>::
compareAndLink( PairNode<Comparab1e> * & first,
                PairNode<Comparab1e> * second ) const
{
      if( second == NULL )
          return;

       if( second->e1ement < first->e1ement )
       {
           // Attach first as leftmost child of second

           second->prev = first->prev;
           first->prev = second;
           first->nextSib1ing = second->leftChild;
           if( first->nextsibling != NULL )
                first->nextSib1ing->prev = first;
           second->leftChi1d = first;
           first = second;
       }
    else
     {
         // Attach second as leftmost child of first
         second->prev = first;
         first->nextSib1ing = second->nextSib1ing;
         if( first->nextSib1ing != NULL )
         first->nextSib1ing->prev = first;
         second->nextSib1ing = first->leftChild;
         if( second->nextSib1ing != NULL )
         second->nextSib1ing->prev = second;
         first->leftChi1d = second;
     }
  }
```

프로그램 12-21. 쌍더미: 두개의 부분더미를 병합하는 루틴

insert와 decreaseKey연산들은 또한 추상적인 서술의 간단한 실현으로 된다. decreaseKey는 위치객체를 요구하는데 이것이 곧 pairNode이다. 이것은 하나의 항목이 처음 삽입될 때 결정되므로 insert는 PairNode에 대한 지적자를 호출측에 되돌려 보낸다. 코드는 프로그람 12-22에 보여 주었다.

```
/**
 * Insert item x into the priority queue, maintaining heap order.
 * Return a pointer to the node containing the new item.
 */
```

```
template <c1ass Comparab1e>
PairNode<Comparab1e> *
PairingHeap<Comparab1e>::insert( const Comparable & x )
{
        PairNode<Comparab1e> *newNode = new PairNode<Comparab1e>( x );
        if( root == NULL )
                root = newNode;
        Else
                 CompareAndLink( root, newNode );
         return newNode;
}
/**
  * Change the value of the item stored in the pairing heap.
  * Does nothing if newVal is larger than currently stored value,
  * p points to a node returned by insert.
  * newVal is the new value, which must be smaller
  * than the currently stored value.
  */
template <c1ass Comparab1e>
void PairingHeap<Comparab1e>::decreaseKey( PairNode<Comparab1e> *p,
                                           const Comparable & newVal )
{
        if( p->e1ement < newVal )
      return;                    // newVal cannot be bigger
        p->element = newVal;
if( p != root )
        {
             if( p->nextSib1ing != NULL )
                 p->nextSib1ing->prev = p->prev;
             if （ p->prev->leftChild == p ）
                  p->prev->leftChild = p->nextSibling;
             else
                  p->prev->nextSib1ing = p->nextSibling;

             p->nextSibling = NULL;
                    compareAndLink( root, p );
         }
}
```

프로그램 12-22. 쌍형더미들: insert 와 decreaseKey

decreaseKey에 대하여 이 루틴은 새 값이 본래의 값보다 더 작지 않으면 즉시에 되돌아 간다. 한편 결과적인 구조는 더미순서를 따르지 않을수 있다. 기본deleteMin절차는 추상적인 서술로부터 직접 유도되며 프로그램 12-23에 보여 주었다.

```
/**
* Remove the smallest item from the priority queue,
* Throws Underflow if empty.
*/
template <c1ass Comparab1e>
void PairingHeap<Comparab1e>::de1eteMin( )
{
    if( isEmpty( ) )
        throw Underf1ow( );
    PairNode<Comparab1e> *oldRoot = root;
    if( root->leftChi1d = = NULL )
        root = NULL;
    else
        root = combineSiblings( root->1eftChild );
    delete oldRoot;
}
```

프로그람 12-23. 쌍더미 deleteMin

구체적으로 보면 CombineSiblingS가 어떻게 실행되는가. 여러가지 변종들이 제안되였으나 피보나치더미와 같은 유도한계를 제공해 주는 변종은 없다. 최근에 제공된 방법들이 리론적으로는 사실상 피보나치더미보다 효과가 적다는것이 밝혀 졌다. 그러나 프로그람 12-24에 작성된 방법은 항상 다른 더미구조들만큼 더 잘 동작하는것처럼 보인다. 다른 더미구조에는 2진더미를 비롯하여 많은 decreaseKey연산을 포함하고 있는 전형적인 그라프리론에 쓰이는 더미들이 들어 간다.

```
/**
 * Internal method that implements two-pass merging.
 * firstSibling is the root of the conglomerate and is assumed not NULL
 */
template    <c1ass    Comparab1e>    PairNode<Comparab1e>    *
PairingHeap<Comparab1e>::
combineSiblings( PairNode<Comparab1e> *firstSibling ) const
{
    if( firstSib1ing->nextSib1ing = = NULL )
          return firstSibling;
             // Allocate the array
    static vector<PairNode<Comparab1e> *> treeArray( 5 );
             // Store the subtrees in an array
    int numSiblings = 0;
    for( ; firstSibling != NULL; numSib1ings++ )
```

```
{
    if( numSiblings == treeArray.size( ) )
        treeArray.resize( nunSiblings * 2 );
    treeArray[ numSiblings ] = firstSibling;
    firstSibling->prev->nextSib1ing = NULL; // break links
    firstSibling = firstSibling->nextSibling;
}
if( numSiblings = = treeArray.size( ) )
     treeArray.resize( numSiblings + 1 );
 treeArray[ numSiblings ] = NULL;

    // Combine subtrees two at a time, going left to right
int i = 0;
for( ; 1 + 1 < numSiblings; i += 2 )
    compareAndLink( treeArray[ i ], treeArray[ i + 1 ] );

int j = i - 2;

  // j has the result of last compareAndLink.
  // If an odd number of trees, get the last one.
 if( j = = numSiblings - 3 )
    compareAndLink( treeArray[ j ], treeArray[ j + 2 ] );

    // Now go right to left, merging last tree with
    // next to last. The result becomes the new last.
for( ; j >= 2; j -= 2 )
    compareAndLink( treeArray[ j - 2 ], treeArray[ j ] );
   return treeArray[ 0 ];
}
```

프로그램 12-24. 쌍더미들: 두 통로병합

2-통로병합(*two-pass-merging*)으로 알려 진 이 방법은 제안된 많은 방법들중에서 가장 간단하고 실용적인 방법이다. 먼저 왼쪽으로부터 오른쪽으로 주사하면서 자식들의 쌍들을 합한다. 첫 주사후에 많은 나무들중에 절반이 병합에 참가한다. 다음 두번째 주사는 오른쪽에서 왼쪽으로 진행된다. 매 단계에서는 첫 주사에서 남은 가장 오른쪽 나무와 현재 합쳐 진 결과와 합친다. 실례로 만일 8개의 자식들 c_1~ c_8을 가지고 있다면 첫번째 주사는 c_1과 c_2, c_3과 c_4, c_5와 c_6, c_7과 c_8을 병합한다. 결과로서 d_1, d_2, d_3, d_4를 얻는다. 두번째에서 d_3과 d_4를 병합한다. d_2는 그 결과와 병합되며 d_1은 그전 병합결과와 합쳐 진다.

이러한것들은 부분나무들을 보관하기 위한 배렬을 요구한다. 최악의 경우에 N-1개의 항목들이 뿌리의 자식으로 될수 있으나(정적이 아닌) 크기가 N인 배렬을 CombineSibling의 안에 선언하면 $O(N)$알고리듬을 얻게 된다. 그래서 단순히 확장한 배렬을 대신 쓴다.

그것이 정적이므로 매 호출때마다 다시 초기화함이 없이 재리용된다.

다른 병합전략들은 련습문제에서 더 준다. 다만 좀 빈약하다고 볼수 있는 간단한 병합전략은 왼쪽-오른쪽 단일통로병합이다(련습문제 12-35). 쌍더미는 《간단한것이 더 좋다.》의 좋은 실례이며 decreaseKey나 병합연산을 요구하는 여러 응용프로그람들에서 선택방법으로 쓰인다.

요약

이 장에서는 2진탐색나무의 여러가지 변종들을 보았다. 내리펼친나무는 유도된 성능 $O(\log N)$을 제공하며 treep는 임의의 성능 $O(\log N)$을 제공하며 흑적나무와 결정성건너뛰기목록, AA-나무모두는 기본조작에 대하여 최악의 경우 실행시간 $O(\log N)$을 준다.
어떤 구조체를 쓰겠는가는 코드의 복잡성과 삭제가 쉬운가 탐색과 삽입시간은 얼마나 적은가에 따라 결심한다. 어느 하나의 구조체가 명백하게 좋다고 말하기는 힘들다. 이미 본 리론들은 나무의 회전과 감시매듭의 리용을 포함함으로써 다른 경우에 필요될수 있는 NULL지시기에 대한 수많은 검사를 제기한다. ***k*차원나무**(*k-d tree*)는 범위탐색에 아주 실천적인 방법을 제공해 주며 리론적인 한계가 최적이 아니여도 된다.

끝으로 쌍더미를 설명하고 코드를 작성하였는데 이것은 특히 decreaseKey연산들이 요구될 때 리론적으로는 피보나치더미보다 효과가 떨어 진다 해도 가장 실천적인 결합가능한 우선권대기렬로 볼수 있다.

련습문제

12-1. 내리펼친구조의 유도된 시간은 $O(\log N)$이라는것을 증명하시오.

12-2. 올리펼친구조에 대하여 호출당 $2\log N$의 회전을 요구하는 대기렬이 존재한다는것을 증명하시오. 내리펼친구조에 대해서 비슷한 결과가 얻어 진다면 증명하시오.

12-3. 펼친나무를 변경시켜 *k*번째 가장 작은 항목에 대한 질문을 반영하도록 하시오. 결정성건너뛰기목록에서 이것이 어떻게 실행되는가.

12-4. 이미 설명된 내리펼친구조와 간단화된 내리펼친구조를 대비하시오.

12-5. 흑적나무에 대하여 삭제과정을 쓰시오.

12-6. 흑적나무의 높이가 최대$2\log N$이며 이 한계를 더는 낮출수 없다는것을 증명하시오.

12-7. 모든 AVL**나무**는 흑적나무처럼 색칠할수 있는가를 알아 보시오. 모든 흑적나

무들이 AVL나무인가?

12-8. 1-2-3결정성건너뛰기목록을 잎과 같은 내부매듭에서의 항목을 가지고 2-3-4나무로써 나타낼수 있는가 알아 보시오.

12-9. 결정성건너뛰기목록에 이미 존재하는 항목을 삽입하려 한다면 어떻게 하면 되는가?

12-10. 1-2-3결정성건너뛰기목록에서 최대로 $2N$개의 매듭이 리용된다는것을 밝히시오.

12-11. C++에서는 매개 추상적인 매듭을 지적자들의 련결목록으로 할대신 동적으로 할당되는 전진방향지적자배렬로 표현할수 있다. 1-2-3결정성건너뛰기도식을 이러한 형식으로 실현하는 방법을 밝히고 매 조작에 대한 $O(\log N)$한계를 보장하시오.

12-12. 1-2-3결정성건너뛰기목록에 대한 삭제절차를 작성하시오.

12-13. AA나무에 대한 삭제알고리듬이 정확하다는것을 증명하시오.

12-14. AA나무를 내리비재귀적형식으로 실현하고 본문에 있는것과 간단한 정도, 효과성정도를 비교하시오.

12-15. Skew와 Split절차를 비재귀적으로 작성하여 매개에 대한 호출이 삭제때 1번씩만 진행되게 하시오.

12-16. BB나무보다 AA나무는 몇행이나 적은 코드를 쓰는가. 이것이 AA나무를 더 빠르게 하는가?

12-17. 탄창을 리용하여 비재귀적으로 트리프에 대한 삽입프로그람을 실현하시오. 품이 더 드는가?

12-18. 호출수를 우선권으로 리용하고 매 호출후 필요될 때 회전을 진행하여 트리프를 자체조절하도록 만들수 있다. 이 방법을 우연적인 방법과 비교하여 보시오. 그리고 항목 X가 호출될 때마다 란수를 발생시키시오. 만일 이 수가 현재항목의 우선권보다 작으면 그것을 X의 새로운 우선권으로 만드시오(적절한 회전을 수행하여).

12-19. 항목들이 정렬되면 트리프를 우선권을 순서지어 주지 않아도 선형시간내에 구성할수 있다는것을 밝히시오.

12-20. nullNode표식을 쓰지 않고 몇가지 나무구조를 실현하시오. 이 표식을 쓰는것에 의하여 코드의 작성품이 얼마나 줄어 드는가?

12-21. 매개 매듭에 대하여 부분나무에 있는 NULL련결들의 수를 보관하고 있다고 해보자. 이것을 매듭의 무게라고 부르자. 다음의 전략을 쓰시오. 만일 오른쪽과 왼쪽 부분나무들이 서로의 차이가 2개 요소이하가 아니라면 완전히 매듭으로부터 시작하는 부분나무를 다시 구성하시오. 다음의것을 밝히시오.

ㄱ. $O(S)$동안에 매듭를 다시 구성할수 있다. 여기서 S는 2매듭의 무게이다.

ㄴ. 알고리듬은 매번 삽입마다 $O(\log N)$의 유도된시간이 든다.

ㄷ. $O(S\log S)$시간동안에 k차원나무에서 매듭을 다시 구성할수 있다. 여기서 S는 그 매듭의 무게이다.

ㄹ. 알고리듬을 k차원나무에 적용하면 삽입당 $O(\log^2 N)$의 시간이 든다.

12-22. 임의의 2차원나무에 대하여 rotatewith Left Chile를 호출한다고 생각해 보자. 결과가 더이상 사용가능한 2진나무가 아니라는 리유를 구체적으로 설명하시오.

12-23. k차원나무에 대하여 삽입과 범위탐색을 진행하시오. 재귀를 쓰지 마시오.

12-24. k=3, 4, 5일 때 이에 대응하는 P의 값에 해당한 부분정합질문을 하는데 걸린 시간을 계산하시오.

12-25. 완전균형k차원나무에 대하여 본문에 설명된 범위질문의 최악의 실행시간을 유도하시오.

12-26. 2차원더미는 매개 항목이 2개의 서로 다른 열쇠를 가지는 자료구조이다. 이 둘중 어느 한 열쇠에 대하여 deleteMin을 시행할수 있다. 2차원더미는 다음의 순서속성을 가지는 완전한 2진나무이다. 짝수깊이의 임의의 매듭 X에 대하여 X에 보관된 항목은 자기의 부분나무에 가장 작은 열쇠 #1을 가지고 있으며 홀수깊이의 임의의 매듭 X에 대하여서는 X에 보관된 항목이 자기의 부분나무에 가장 적은 열쇠 #2를 가지고 있다.

ㄱ. 항목 (1.10),(2.9),(3.8),(4.7),(5.6)에 대하여 가능한 2차원더미를 그리시오.

ㄴ. 최소열쇠 #1을 가지는 항목을 어떻게 찾는가?

ㄷ. 최소열쇠 #2를 가지는 항목은 어떻게 찾는가?

ㄹ. 2차원더미에 새 항목을 삽입하는 알고리듬을 작성하시오.

ㅁ. 어느 한 열쇠에 대하여 deleteMin을 수행하는 알고리듬을 작하시오.

ㅂ. 선형시간동안에 buildHeap를 수행하는 알고리듬을 작성하시오.

12-27. k차더미를 얻기 위한 우의 련습문제를 일반화하시오. 이 더미에는 k개의 개별적열쇠가 있다. 다음의 한계를 얻을수 있다. 삽입은 $O(\log N)$, 삭제는 $O(2^k \log N)$, buildHeap는 $O(kN)$의 시간이 걸린다.

12-28. 2차원더미를 써서 쌍방향을 가지는 대기렬을 실현할수 있다는것을 밝히시오.

12-29. 추상적으로 k차원더미를 일반화하여 열쇠 #1에서 갈라 지는 준위들만 2개의 자식들을 가지도록 하시오(다른것들은 하나).

ㄱ. 지적자가 필요한가?

ㄴ. 명백히 기초알고리듬도 아직 동작한다. 새로운 시간한계는 얼마인가.

12-30. *k*차원나무를 써서 deleteMin을 실현하시오. 우연나무에 대하여 평균가동시간을 얼마로 기대하게 되는가?

12-31. *k*차원더미를 써서 역시 deleteMin을 제공하는 쌍방향대기렬을 실현하시오.

12-32. nullNode표식을 가지고 쌍더미를 실현하시오.

12-33. 본문의 쌍더미알고리듬에 대하여 매개 연산의 유도된 시간이 $O(\log N)$임을 밝히시오.

12-34. CombineSiblings에 대하여 다른 방법은 모든 형제들을 대기렬에 놓는것이다. 그리고 반복적으로 쌍방향대기렬을 실행하고 대기렬의 첫 두 항목을 합한 다음 합한 결과를 대기렬의 맨 끝에 놓는것이다. 이 변형을 실현하시오.

12-35. 앞련습에서 대기렬대신에 탄창을 쓰는것이 연산당 $O(N)$의 시간이 들게 하는 순서렬을 쓰기때문에 나쁘다는것을 밝히시오. 이것이 왼쪽-오른쪽 단일통로병합이다.

12-36. decreasekey를 쓰지 않고서도 부모련결을 제거할수 있다. 결과를 경사더미와 비교해 보면 얼마나 좋은가.

12-37. 다음의 매개가 자식과 부모지적자를 가지는 나무로써 표현된다고 본다. decreaseKey연산을 실현하는 방법을 설명하시오.

ㄱ. 2진더미

ㄴ. 펼친나무

12-38. 그라프적으로 보면 2차원나무에서 매개 매듭은 평면을 나누는데 두개 구역으로 분할한다. 실례로 그림 12-28에 그림 12-24의 2차원나무에 5개를 처음으로 삽입한것을 보여 주었다. *p*1의 첫 삽입은 평면을 왼쪽과 오른쪽으로 나눈다. *p*2의 두번째 삽입은 왼쪽 부분을 우와 아래로 나눈다. 이렇게 계속된다.

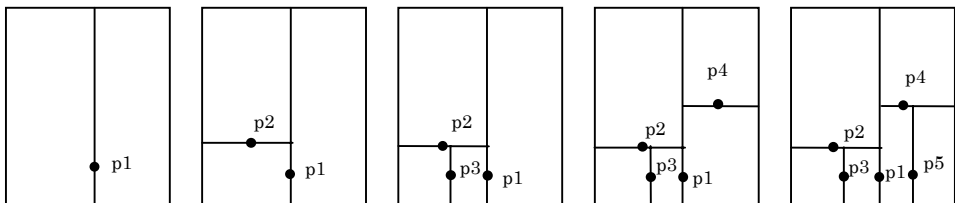



그림 12-28. P_1, P_2, P_3, P_4, P_5의 삽입후 2차원나무에 의하여 갈라 진 평면

ㄱ. 주어 진 N개의 항목들의 모임에 대하여 삽입순서가 최종부분에 영향을 미치는가.

ㄴ. 2개의 서로 다른 삽입순서렬이 같은 나무로 된다면 같은 구역들이 생성되는가.

ㄷ. N개의 삽입후 생기는 구역들의 수에 대한 공식을 끌어 내시오

ㄹ. 그림 12-24의 최종구역을 밝혀 내시오. 공식을 끌어 내시오.

12-39. 2차원나무를 대신할수 있는것이 4진나무이다. 그림 12-29에 4진나무에 의해 평면이 어떻게 나누어 지는가를 보여 주었다. 초기에는 (바른4각형인데 꼭 그럴 필요는 없다.)하나의 구역을 가지고 있다. 매개 구역은 1개의 점을 보관한다. 만일 두번째 점이 삽입되면 그 구역은 4개의 같은 크기의 4각형으로 (북동, 남동, 남서, 북서)나누인다. 이것을 서로 다른 4각형에 점들을 놓는다면 ($p2$이 삽입될 때)다 된것으로 된다. 다시말하여 재귀적으로 분리를 진행할수 있다($p5$이 삽입될 때까지).

ㄱ. 주어 진 N개의 항목들의 모임에 대하여 삽입순서가 최종구역에 영향을 미치는가.

ㄴ. 그림 12-24와 같은 2진나무에 있던 같은 요소들이 4진나무에 삽입될 때의 최종구역을 그려 보시오.

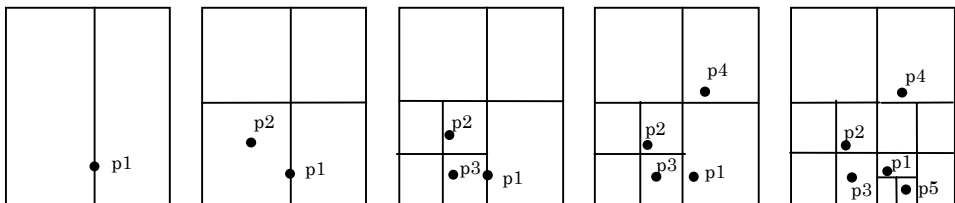



그림 12-29. P_1, P_2, P_3, P_4, P_5의 삽입후 4 진나무로 갈라 진 평면

12-40. 나무자료구조에는 4진나무를 보관할수 있다. 원래의 구역의 한계를 보관한다. 나무의 뿌리는 원래구역을 나타낸다. 매개 매듭은 삽입되는 매듭을 보관하고 있는 잎 또는 4개의 4각형을 나타내는 4개의 자식을 가지고 있다. 탐색을 위하여 뿌리로부터 시작하여 잎(또는 NULL항목)에 도달할 때까지 적절한 4각형으로 반복적으로 가지쳐 간다.

ㄱ. 그림 12-29에 해당한 4진나무를 그리시오.

ㄴ. 4진나무가 깊어 지는데 영향을 주는것은 어떤 요소인가.

ㄷ. 4진나무에 대하여 직교범위질문을 하는 알고리듬을 작성하시오.

참고문헌

내리펼친나무들은 원래의 펼친나무에 대한 서술에서 설명하였다.위험한 회전을 없앤 류사한 방안을 [31]에서 설명하였다. 내리흑적나무알고리듬은 [17]에서 보여 준다. 보다 읽기가능한 설명은 [28]에서 구체적으로 주었다. 감시매듭을 쓰지 않은 내리흑적나무의 실현을 [14]에 주고 있다. 이것은 nullNode의 효과성을 보여 준다. 결정성건너뛰기목록과 그의 변종들을 [23]과 [26]에서 설명한다. 대칭2진B나무들은 [6]에 주었다. 본문에서 보여 준 AA나무를 실현한것은 [1], [3]이다. 트리프[4]들은 [32]에서 서술한 **데카르트나무**(*Cartesian*)에 기초하고 있다. 련관된 자료구조는 우선권탐색나무[21]에서 주었다.

***k*차원나무**는 [7]에서 처음 서술하였다. 다른 범위탐색알고리듬들은 [8]에서 설명하였다. K차원균형나무에서 범위탐색의 최악의 경우는 [19]에서 얻어 졌으며 평균경우와 결과들은 [13]과 [10]에서 얻어 진것들이다.

쌍더미와 련습문제에서 제기된 치환물들은 [16]에 서술되였다. 연구결과 [18]는 펼친나무가 decreaseKey연산이 요구되지 않는때와 선택의 우선권렬이라는것을 제기한다. 다른연구 [30]은 쌍더미가 피보나치더미와 같은 근사적인 한계를 가지며 성능은 더 실천적으로 더 좋다. 그러나 우선권대기렬을 써서 최소나무순회알고리듬을 실현하는 련관된 연구 [22]는 decreaseKey의 유도시간 $O(1)$이라는것을 제기한다. M.Friedman[15]은 decreasekey의 유도된 비용이 부분최적(실지로 적어도 $\Omega(\log\log N)$)인 대기렬이 있다는 증명으로부터 서술할수 있다. 다른 한편 그는 또한 prim의 최소생성나무(minimum spanning tree)알고리듬서술에 리용하면 그라프가 약간 조밀한 경우에(즉 그라프에서 임의의 ε에 대하여 정점수가 $O(N^{(H\varepsilon)})$이다.) 최적이라는것을 보여 주고 있다. 그러나 쌍더미의 분석은 여전히 남아 있다.

련습문제의 모든 풀이는 먼저 참고문헌을 보아야 구할수 있다. 련습문제 12-21은 많이 리용하고 있는《지연》균형전략을 참고한다. [20], [5], [11], [9]는 특이한 전략을 서술한다. 즉 [2]는 하나의 체계안에서 이 모든 전략들을 어떻게 실현하는가를 보여 준다. 련습문제 12-21에서 특성을 렬거한 나무는 무게균형나무(weight-balanced) 이다. 이나무들은 또한 [24] 회전에 의해서 유지될수 있다. 련습문제 12-26부터 12-28사이의 련습문제들에 대한 풀이를 [12]에서 줄수 있다. 4각형나무는 [27]에서 서술되였다.

1. A. Anderson, "A Note on Searching a Binary Search Tree," *Software—Practice and Experience,*21 (1991), 1125-1128.
2. A. Andersson, "General Balanced Trees," *Journal of Algorithms,* to appear.
3. A. Andersson, "Balanced Search Trees Made Simple," *Proceedings on the Third Workshop on algorithms and Data Structures* (1993), 61-71.
4. C. Aragon and R. Seidel, "Randomized Search Trees," *Proceedings of the Thirtieth Annual Symposium on Foundations of Computer Science* {1989}, 540-545.
5. J. L. Raer and B. Schwah, "A Comparison of Tree-Balancing Algorithms," *Communications of the ACM,* 20(1977), 322-330.
6. R. Bayer, "Symmetric Binary B-Trees: Data Structure and Maintenance Algorithms," *Acta* Informatica, I (1972), 290-306.
7. J. L. Benrley, "Multidimensional Binary Search Trees Used for Associative Searching," *Communications of the ACM,* 18 (1975), 509-517.
8. S. J. L. Bentley and J. H. Friedman, "Data Structures for Range Searching," *Computing Surveys,* 11 (1979), 397-409.
9. H. Chang and S. S. lyengar, "Efficient Algorithms to Globally Balance a Binary Search Tree," *Communications of the* ACM, 27 (1984), 695-702.
10. P. Chanzy, "Range Search and Nearest Neighbor Search," *Master's Thesis,* McGill University (1993).
11. A. C. Day, "Balancing a Binary Tree,"- *Computer journal,*19 (1976), 360-361.
12. Y. Ding and M. A. Weiss, "The k-d Heap: An Efficient Multi-Dimensional Priority Queue," *Proceedings of the Third Workshop on Algorithms and Data Structures* (1993), 302-313.
13. P. Flajolct and C. Puech, "Partial Match Retrieval of Multidimensional Data," *Journal of the ACM,* 33 (1986), 371-407.
14. B. Flaming, *Practical Data Structures in* C++, John Wiley, New York (1994).
15. M. Friedman, "Information Theoretic Implications for Pairing Heaps," *Proceedings of the Thirtieth Annual* ACM *Symposium on the Theory of Computing,* (1998) 319-326.
16. M. L Friedman, R. Sedgewick, D. D. Sleator, and R. E. Tarjan, "The Pairing Heap: A New Form of Self-Adjusting Heap," *Algorithmica,* I (1986), 111-129.
17. L. J. Guibas and R. Sedgewick, "A Dichromatic Framework for Balanced Trees," *Proceedings of the Nineteenth Annual Symposium on Foundations of Computer Science* (1978), 8-21.

18. D. W. Jones, "An Empirical Comparison of Priority-Queue and Event-Set Implementations," *Communications of the ACM,* 29 (1986), 300-311.
19. D. T. Fee and C. K. Wong, "Worst-Case Analysis for Region and Partial Region Searches in Multidimensional Binary Search Trees and Balanced Quad Trees," *Acta Informatica,* 9 (1977), 23-29.
20. W. A. Martin and D. N. Ness, "Optimizing Binary Trees Grown with a Sorting Algorithm," *Communications of the ACM,* 15 (1972), 88-93.
21. E.McCreighr, "Priority Search Trees," *Sl AM journal of Computing,* 14 (1985), 257-276.
22. B. M. E. Moret and H. D. Shapiro, "An Empirical Analysis of Algorithms for Constructing a Minimum Spanning Tree," *Proceedings of the Second Workshop on Algorithms and Data Structures,* (1991), 400-411.
23. J. I. Munro, T. Papadakis, and R. Sedgewick, "Deterministic Skip Eists," *Proceedings* of-*the Third Annual Symposium of Discrete Algorithms* (1992), 367-375.
24. J. Nievergelt and E. M. Reingold, "Binary Search Trees of Bounded Balance," 511-*Journal on Computing,* 1 (1973), 33-43.
25. M. H. Overmars andJ. van Eeeuwen, "Dynamic Multidimensional Data Structure.S lwel on Quad and K-D Trees," *Acta Informatica,* 17 (1982), 267-285.
26. T. Papadakis, *Skip Lists and Probabilistic Analysis of Algorithms,* Ph.D. University of Waterloo (1993).
27. H. Samer, "The Quadtree and Related Hierarchical Data Structures," *Computissertation* 16(1984), 187-260.
28. R. Sedgewick, *Algorithms in* C++, Addison-Wesley, Reading, Mass. (1992),
29. D. D. Sleator and R. E. Tarjan, "Self Adjusting Binary Search Trees," *Journal* of the ACM 32(1985), 52-686.
30. J. T. Stasko andJ. S. Vitter, "Pairing Heaps: Experiments and Analysis," *Communications of the ACM,* 30 (1987), 234-249.
31. C. J. Stephenson, "A Method for Constructing Binary Search Trees by iVIaking Insertions at the Root," *International Journal of Computer and Information Science, 9* (1980), 15-29.
32. J. Vuillemin, "A Unifying Look at Data Structures," *Communications of the* A CM, 23 (1980), 229-239.

부록 A. 표준형판서고

지금 발표되는 C++표준은 모든 프로그람실행을 **표준형판서고**(*standard Template Library*)(혹은 간단히 STL) 와 같은 서고를 써서 할것을 요구하고 있다. STL은 자료구조(목록, 탄창, 대기렬, 우선권대기렬과 같은)와 알고리듬(정렬, 선택과 같은)의 모임으로 되여 있다. 이름그대로 STL에서는 현재의 많은 콤파일러(이 책에서는 론의하지 않는다.)에서는 동작하지 않는 개선된 특성을 가진 형판들을 많이 리용하고 있다. 결론적으로 지금 상황에서 보면 STL이 정확히 실행된다는 확신은 있지만 STL의 실행이 완전히 완성되였다고 볼수는 없다. 앞에서 학습한 많은 개념들을 STL을 리용해 보는것으로 더욱 확고히 인식할수 있으므로 STL리용은 매우 흥미 있는 문제로 된다. 또한 앞에서 본 자료구조들이 비록 기본적인 방법이기는 하지만 STL과 같은 더 완전한 묶음을 리용하는것과 아주 류사하다는것을 보게 된다.

부록 A에서는 다음의 문제를 취급하려고 한다.

- STL의 조직서술, 프로그람의 서술없이 STL의 통합.
- STL의 목록과 모임, 배치표고찰.
- STL을 쓴 두개의 C++실례프로그람에서 무게없는최단경로를 계산하는 알고리듬을 실행.
- 두개의 C++프로그람들은 제1장부터 제12장까지에서 개발된 자료구조클라스들을 써서 재실행.

부록 1. 소개

STL은 앞에서 서술해 온 몇개의 자료구조들의 실행을 포함하고 있는데 여기에는 특히 련관된 반복자를 가진 쌍방향련결목록과 우선권대기렬, 균형탐색나무를 리용하는 자료구조를 가진 련결된 목록클라스가 있다. 이 클라스들의 특성은 앞에서 서술한 클라스와는 좀 차이가 있지만 개념과 알고리듬, 실행시간은 같다. STL은 하쉬표자료구조나 단일/탐색자료구조를 제공하지 않는다. STL에는 2진탐색알고리듬과 고속정렬알고리듬이 있다.

STL은 C++서고의 일부분으로서 방대한 검사와 최적화를 진행하는데 아주 적합하며 수많은 프로그람작성전문가들에 의해 널리 리용되고 있다. 그러므로 일반적으로 다른 실행보다는 차라리 이 서고를 리용하는 편이 퍽 낫다. 이 책에서는 STL의 완벽한 적용범위를 주었다.

부록 2. STL의 기본개념

여기서는 STL의 기초적인 문제를 보게 된다. 즉 새로운 머리부파일과 using명령, 용기와 반복자, 쌍(pairs)들과 기능객체에 대하여 보게 된다.

1. 머리부파일과 using명령

전통적으로 코드작성방식에서는 서고의 **머리부파일**(*header file*)이름들이 뒤붙이 *.h로 끝난다. 새로 표준으로 이름 지어 준 지령은 뒤붙이가 없다. 실례로 표준 I/O머리부파일은 iostream.h대신 iostream으로 쓴다. 많은 실행에서는 여전히 iostream.h머리부파일을 제공할것을 요구한다. 여기서 보면 STL의 머리부와 우의 머리부파일은 일치하지 않는다. Visual C++ 5.0에서 실례로 STL의 임의의 머리부파일을 리용하려고 한다면 iostream.h라고 쓸수 없다. 몇가지 다른 머리부파일들은 fstream, sstream, vector, list, deque, set, map이다. 새로 발표된 표준은 새로운 기능 즉 namespace로 불리워 지는 기능을 더 추가한것이다. 비록 이름쓰기가 중요하다고 해도 여기서는 그것들의 리용방법을 취급하지 않는다. 여기서 중요한것은 전체 STL이름대역에서 정의되여야 한다는것이다. 가령 공동정의부에 그것이 정의되여 있다면 STL을 호출하기 위해 프로그람 A-1에서와 같이 **using지령** *(using directive)*을 쓸수 있다.

Using namespace std;

현재의 C++책들에 다른 방법들도 있다고 해도 우의 방법이 그중 간단하다. 프로그람 A-1에서는 새로운 머리부파일 iostream과 using명령에 대하여 설명한다.

```
# include <iostream>
using namespace std;
int main( )
{
        cout << “First program” << endl;
        return 0;
}
```

프로그람 A-1. 새로 STL을 쓴 첫 프로그람

2. 용기

용기(*containers*)는 대상들을 요소로 가지는 대상의 모임이다. 벡토르와 목록과 같은 실행들은 순서화되지 않았으며 모임과 배치표와 같은 다른 실행들은 순서화되였다. 일부

실행들은 중복되는것을 허용하지만 또 중복을 허용하지 않는것도 있다. 모든 용기들은 아래와 같은 조작을 할수 있다.

bool empty( ) const

용기에 요소들을 포함하지 않는다면 론리값 true, 포함하면 false를 준다.

iterator begin( ) const

용기에서 모든 대상의 위치를 움직여 가도록 하는데 리용될수 있는 반복자를 준다.

iterator end( ) const

끝표식이 붙은 즉 용기에서 마지막요소의 위치를 지적하는 반복자를 준다.

int size( ) const

용기에서 요소들의 수를 준다.

이 방법들에서 제일 중요한것은 반복자를 주는것이다. 이 조작은 부록 2의 세번째에서 구체적으로 서술한다.

3. 반복자

이름에서 보는바와 같이 **반복자**(*iterator*)는 그룹에 들어 있는 모든 객체들에 대하여 어떤 조작을 반복하게 하는 하나의 객체이다. 반복자객체을 리용하는 기술은 제3장의 련결목록부분에서 서술되였다. STL반복자는 제3장의 련결목록부분에서 서술한 반복자와 대체로 같은 개념으로 리용하지만 언어처리기의 서고에서 기대하는것만큼 더 강력한 기능을 부여해 준다.

실지로 반복자의 형은 여러가지이다. 어느것이든 모든 반복자형에 대하여 변화되는 조작에 대하여 셀수 있다는것이다.

itr++

이 반복자 itr를 다음 걸음으로 이행하게 한다. 앞으로 전진, 뒤로 후진하는 두가지 형이 가능하며 되돌림형은 반복자의 형으로 규정된다.

*itr

이 조작은 반복자 itr위치에 기억된 대상의 주소를 준다. 주소는 변경할수 있는 혹은 변경할수 없는것일수 있는데 이것은 반복자의 형에 관계된다. 실례로 const_iterator는 const의 용기를 움직이게 하는데 리용하는데 const의 주소를 되돌려 주는 operator*를 가진다. 이로부터 값주기명령문의 왼쪽 변에 *itr를 쓸수 없다.

itr1==itr2

이 조작은 반복자 itr1과 itr2가 같은 주소에 있다면 론리값 true, 아니면 False를 준다.

itr1!=itr2

이 조작은 반복자 itr1과 itr2가 다른 주소에 있다면 true, 아니면 false를 준다.

매개의 용기는 여러개의 반복자를 정의한다. 실례로 list<int>는 list<int>∷ iterator와 list<int>∷const_iterator로 정의할수 있다. const_iterator는 용기가 변경될수 없는 용기라면 iterator대신에 쓸수 있다.

실례에서 보는것처럼 프로그람 A-2에서는 임의의 용기에 있는 매개 요소들을 출력하였는데 요소는 이를 위하여 정의된 연산자 << 를 가진다. 용기가 순서화된 모임이라면 이 요소들은 정렬된 순서대로 출력된다.

```
// Print the contents of Container c
template <c1ass Containers>
void printCo11ection( const Container & c )
{
      Container::const_iterator itr;
      for( itr = c.begin( ); itr != c.end( ); itr++ )
            cout << *itr << '\ n';
}
```

프로그람 A-2. 용기의 내용출력

4. 쌍

단순한 실체에서 객체의 **쌍**(*pair*)을 기억시켜야 할 필요가 자주 제기된다. 이것은 동시에 두개의 실체를 되돌려 주어야 할 때 유효하다. 또한 부록 A의 5에서 이야기하려는 map클라스에서도 리용된다. STL은 아래와 같은 의미를 가지는 하나의 클라스형판 pair를 정의한다.

```
template <c1ass Object1, class Object2>
class Pair
{
     public:
            Object1 first;
            Object2 second;
};
```

5. 함수객체

일반적으로 순서짓기특성을 요구하는 용기의 알고리듬은 주로 지정된 순서(일반적으로 less는 객체연산자 <를 호출하는것과 같은 처리를 진행)를 가진다.

이 알고리듬은 주로 여러가지 순서짓기속성을 특징 지어 주는 기능을 수행한다. 이 알고리듬은 순서대로 정렬하려는 대상이 요구하는대로 정렬되여 있지 않을 때 아주 유용하다. 실례로 기호렬로 된 벡토르를 정렬하려고 할 때 혹은 길이에 따라 기호렬 등을 정렬하려고 할 때 아주 효과적이다.

이 실례를 프로그람 A-3에서 보여 주었다. comp는 길이에 따라 기호렬들을 비교한다. 이 기능은 대상을 형태적으로 정렬하는 세번째 인수의 선택적인 리용에 의하여 수행된다. 기능객체는 이 조작자()를 위한 실행으로 정의되는데 이것이 기능호출조작자이다. 이때 분류를 위하여 기능객체대신 세번째 인수를 리용한다.

기능객체가 자료성원과 구축자(constructor)가 아닌것을 포함하더라도 보다 일반기능객체들은 가능하다. 다만 조작자()가 정의되여야 한다. STL은 less(많은 용기알고리듬에 정의되여 있는)와 greater라는 많은 형판기능객체를 포함하고 있다.

```
class Comp
    {
        public:
            bool operator( ) ( const string & 1hs,
                                        const string & rhs ) const
            { return lhs.length( ) < rhs.1ength( ); }
    };
    void sortListOfStringsByLength( vector<string> & array )
    {
          sort( array.begin( ), array.end( ), Comp( ) );
    }
```

프로그람 A-3. 기능객체를 리용한 정렬알고리듬

부록 3. 비순서렬: 벡토르와 목록

벡토르와 목록은 둘다 비순서화된 용기(또는 대기렬로 알려 졌다.)의 실행에 쓰인다. 리용자는 대기렬에서 매개의 요소를 삽입하는 명확한 조작을 할수 있다. 사용자는 대기렬에서 그 요소들의 위치에 따라 요소들을 호출할수 있으며 대기렬에서 요소들에 대한 탐색을 진행할수 있다. 이것들은 다 특이한 조작에 의거하지만 벡토르나 목록에서는 아주 능률적이다.

1. 벡트로와 목록

STL은 3개의 선차적인 실행을 제공한다. 여기서 2개는 다만 일반적인 형태로서 배렬에 기초한 변종(version)과 쌍사슬목록(doubly linked list)에 기초한 변종이다. 배렬에 기초한 변종은 제3장에서 서술한것처럼 배렬의 제일 끝에서만 삽입하려고 할 때 아주 적합한것이다. STL은 배렬의 제일 끝에 삽입하려고 할 때 내부능력을 넘어 서서 묶음은 배로 된다. 이것이 비록 훌륭한 큰O표기법의 수행결과를 가져다 준다고 하더라도 목록은 구조적으로 확장된 큰 객체에 대하여 구축자(constructors)에 대한 호출을 최소로 하는데 아주 적합하다.

순서렬도중에서의 삽입과 삭제는 벡토르에서 사실 능률적이지 못하다. 벡토르는 색인으로 직접 호출할수 있지만 목록은 그렇지 못하다. 그래서 목록은 항상 색인을 필요로 하지 않는데서 쓰기가 매우 쉽다. 벡토르는 삽입이 끝에서만 진행되며 삽입된 대상이 구조적으로 크게 확장되지 못하더라도 탐색에서는 괜찮다. 순서렬에 대하여 아래와 같은 보충적인 조작을 할수 있다.

void push_back(const object element)

이 순서렬의 끝에 요소를 추가한다.

void push_front(const ovject & element)

이 순서렬의 맨 선두에 요소를 추가한다. 벡토르에 대하여서는 변화시키기가 매우 어렵기때문에 대체로 변화시키지 않는다. 량끝대기렬(deque)은 벡토르와 류사하게 작용하지만 량끝호출이 가능하다.

object & front() const

순서렬에서 첫 요소를 되돌려 준다.

object & back() const

순서렬에서 마지막요소를 되돌려 준다.

void pop_front( )

순서렬에서 첫 요소를 제거한다. 이것은 목록과 량끝대기렬에서만 가능하다.

void pop_back( )

순서렬에서 마지막요소를 제거한다.

iterator insert (iterator pos,const object & obj)

대상 obj를 pos가 지적하는 위치요소의 바로 앞에 삽입한다. 이 조작은 목록에 대해

서는 항상 가능하며 그러나 벡토르에서는 순서렬의 pos로부터 끝까지사이의 거리에 대응할 때 취할수 있다. 새롭게 삽입된 항목의 위치를 준다.

iterator erase(iterator pos)

pos로 지적된 위치에 있는 객체를 제거한다. 대기렬에서 요소들은 요구된대로 론리적으로 제거한다. 이 조작은 List에서는 상수시간이 걸리나 벡토르에서는 순서렬의 pos위치로부터 끝까지 거리에 비례되는 시간이 걸린다. 우의 명령은 erase를 호출하기전의 pos에 들어 있는 요소의 위치를 되돌려 준다.

실례로 프로그람 A-4에서는 표준입력과 출력으로부터 정렬된 순서대로 옹근수들을 읽어 내는 프로그람을 보여 주고 있다. 벡토르가 구축될 때 그것은 비여 있든지 아니면 크기가 초기화되여야 한다는데 주의를 준다. 벡토르에서 요소들은 적합한 초기값을 가지고 초기화된다. 그러므로 벡토르는 작업하려고 하는 지금 상황에서 크기 0으로 초기화되여야 한다.

```
# inc1ude <iostream>
# include <vector>
# include <a1gorithm>
using namespace std;

 int main( )
 {
        vector<int> v ;            // Initial size is 0
        int x ;

        whi1e( cin >> x )
              v.push_back( x );

        sort( v,begin( ), v.end( ) );

        for( int i = 0; i < v.size( ); i++ )
              cout << v[ i ] << endl;

        return 0;
 }
```

프로그람 A-4. 벡토르에서 push back 의 리용

2. 탄창과 대기렬

STL은 탄창과 대기렬클라스를 제공하지만 이것들은 해당 기능들을 호출하는 렬용기(목록, 벡토르 즉 량끝대기렬)를 쉽게 리용한다.

대기렬은 enqueue와 dequeue와 같은 표준이름을 리용하지 않는다. 그러므로 렬용기를 직접 리용하지 않도록 강요할 까닭이 없다. 프로그람 A-5는 대기렬클라스를 list클라스에 포함시키는것이 어렵지 않다는것을 례증하고 있다.

```
# inc1ude <1ist>
using namespace std;

template <c1ass Object>
class Queue
{
    public:
        bool isEmpty( ) const;
        const Object & getFront( ) const;
        void makeEmpty( );
        Object dequeue( );
        void enqueue( const Object & x );
    private:
        list<Object> theList;
};
template <c1ass Object>
bool Queue<Object>::isEmpty( ) const
{
        return theList.empty( );
}
template <c1ass Object>
const Object & Queue<Object>::getFront( ) const
{
    if( isEmpty( ) )
        throw Underflow( );
    return theList.front( );
}
template <c1ass Object>
void Queue<Object>::makeEmpty( )
{
whi1e( ! isEmpty( ) )
        dequeue( );
}
template <c1ass Object>
Object Queue<Object>::dequeue( )
{
    Object frontItem = getFront( );
```

```
        theList.pop_front( );
        return frontItem;
  }
  template <c1ass Object>
  void Queue<0bject>::enqueue( const Object & x )
  {
        theList.push_back( x );
  }
```

프로그람 A-5. STL 목록을 써서
집행된 대기렬클라스

부록 4. 모임

모임(*set*)은 순서화된 용기이다. 모임에서 요소들은 중복되지 않을것을 요구한다(일부 다중모임은 요소중복이 가능하지만 다중모임은 여기서 취급하지 않는다.). 아래에서 서술하는 실행코드는 평형이 갖추어 진 탐색나무이다. 모임은 보통 begin, end, size 그리고 empty를 리용하는데 추가적으로 다음의 조작(연산)들도 모임에서 리용된다.

pair<iterator, bool>insert(const object & element)

만일 모임에 요소가 존재하지 않는다면 element를 모임에 추가한다. Bool인수는 모임에 요소가 이미 포함되여 있지 않다면 true라는 론리값을 되돌려 준다. 포함되여 있으면 false라는 론리값을 준다. iterator요소는 모임에 요소가 위치되여 있는 위치값을 되돌려 준다.

iterator find(const object & element)const

모임에서 요소의 위치값 iterator를 준다. 또는 모임에 요소가 없으면 end()를 준다.

int erase (const object & element)

만일 존재하면 모임에서 요소를 제거한다. 우의 명령은 제거되는 요소들의 수를 되돌려 주는 명령이다(이로부터 0아니면 1이다.)

보통 순서화는 less<object>라는 함수객체를 리용하는데 이 대상은 또 자체가 object에 대하여 연산자<를 호출함으로써 실행되게 된다. 다음 한가지 순서화는 함수객체형인 set 형판을 써서 렬거할수 있다. 실례로 프로그람 A-6에서 기호렬로 된 모임이 어떻게 구조화되는가를 보여 주고 있다. print collection을 호출하면 작아 지는 순서대로 출력하게 된다.

```
# inc1ude <iostream>
# include <set>
# inc1ude <string>
using namespace std;
int main( )
{
        set<string, greater<string> > s;            // Use reverse order
        s.insert( "Joe" );
        s.insert( "bob" );
        printCo11ection( s );                        //  그림  A-2
        return 0;
}
```

프로그람 A-6. 반대순서로 배렬된 모임의 실례

부록 5. 배치표(map)

배치표(*map*)는 열쇠(key)들과 열쇠의 값(Value)들로 구성되여 있는 순서 있는 자료입력렬들의 결합을 기억시키려고 할 때 리용된다. 배치표는 값에 따라 배치한다. 열쇠들은 유일해야 하며 그러나 여러개의 열쇠들은 같은 값들을 가질수 있다. 그러므로 값은 유일하지 않을수도 있다.

배치표는 탐색시간이 로그적으로 계산되는 평형이 갖추어 진 탐색나무를 리용한다. 배치표는 쌍 지은 모임처럼 작용하는데 이것들은 열쇠만을 참고하는 비교기능을 가지고 있다.[34] 그래서 이것은 begin, end, size와 empty를 지원하며 그러나 아래에서 보는것처럼 반복자는 열쇠-값의 쌍으로 되여 있다. 다른 말로 표현하면 반복자 itr, *itr는 pair<열쇠형, 값형>형이다. 배치표는 삽입과 탐색, 삭제연산을 할수 있게 하는 대상이다. 삽입을 위한 한가지 방법은 pair<열쇠형,값형>객체를 리용하는것이다. find가 열쇠만 요구해도 반복자는 열쇠-값 쌍을 참고한 값을 되돌려 준다.

배렬의 첨수화조작은 배치표에 포함되여 있다. 여기서는 non-const와 const의 변종 두가지에 대한 기능정의를 한다.

```
value Type & Operator[ ](const key Type & key)
Const Value Type & Operator[ ](Const Key Type & key)Const
```

이 매개는 열쇠가 배치표에 의해 배치되는 값을 되돌려 준다. 열쇠가 배치되지 않았으면 열쇠는 령인 수구축자를 동작시킴으로써 얻어 진 지정된 Value Type에 배치된다.

[34] 다중모임은 열쇠가 중복될수 있으나 여기서는 다중모임을 취급하지 않는다.

이 문법형태는 때로는 관계배렬이라고도 알려 져 있다. 배치표실례에서 간단히 본것처럼 프로그람이 몇개의 형으로 구성되였다. 프로그람 A-7에서 People배치표는 string을 int에 배치한다. 하여 Tim은 초기에 3이며 다음에는 5이다. 이 값들은 첫 명령문에 의하여 출력된다. Bob는 출력명령문앞의 map에는 없지만 Operator[ ]를 호출함으로써 map에 초기값 0을 가지게 된다. 그래서 0이 두번째 출력명령문에 의하여 출력(아마 계획적으로)된다. Bob가 배치표에 있는가 하는것을 알자면 findfirst를 호출해 보아야 하며 또한 반복자가 end()와 같은 값을 되돌려 주는가를 검색해야 한다.

```
 # inc1ude <iostream>
# inc1ude <map>
# inc1ude <string>
using namespace std;

int main( )
{
      map<string, int> people;

      peop1e[“Tim”] = 3; peop1e["Tim"] = 5;
      cout << "Tim's value is " << peop1e["Tim"] << endl;
      cout << "Bob's value is " << peop1e["Bob"] << endl;

      return 0:
 }
```

프로그람 A-7. 배치표실례: Tims 의 값은 5, Bob 의 값은 0

일단 find를 호출하면 반복자 itr가 값을 가지게 되며 그 값을 탐색하기 위하여 두번째 탐색을 피하도록 하는데 itr->second를 리용할수 있다.

부록 6. 실례: 색인만들기

한개 파일의 용어**색인**(*concordance*)은 단어가 출현하는 행의 번호를 가지고 파일의 모든 단어들을 포함하는 목록렬이다. STL을 써서 우리는 색인을 만드는 프로그람을 작성할수 있다. 하나의 단어가 출력되지 않는 공백기호도 포함하는 련속적인 임의의 기호렬이라고 가정하자. STL을 써서 색인을 만들어 내는 프로그람을 작성할수 있다.

기본사상은 단어가 출현하는 행의 목록에 단어들을 배치하며 배치표를 리용하는것이다. 그러나 매개 열쇠는 하나의 단어이며 그 값은 행번호들로 된 목록이다. 하나의 단어를 보면 그것이 배치표에 이미 배치되였는가 하는 검사를 한다. 만일 그것이 있으면 그 단어와 일치하는 목록의 현재행번호를 간단히 추가한다. 그것이 없으면 현재행번호가

포함되여 있는 목록과 함께 그에 속하는 단어를 배치표에 추가한다. 모든 단어들을 읽은 다음 배치표를 참고하면서 이 공정을 반복한다. 이것은 열쇠건에 따라 정렬된 순서대로 배치표를 기입하며 정렬된 순서대로 출현하게 된다. 매개의 배치표의 기입을 위하여 단어들을 출력하고 다음행번호의 련결목록에 가서 그것들을 출력한다.

1. STL방안

STL을 리용한 프로그람을 프로그람 A-8에 보여 주었다. 먼저 main을 고찰해 보자. main에서 하나의 파일을 열고 배치표를 만든다. 행번호를 기억시키는데 벡토르나 목록을 리용할수 있으므로 두개 자료구조에 대하여 기능이 좋은 push_back연산이 가능하다. 첫 for순환에서 현재행번호를 보존하면서 한행을 단번에 반복적으로 읽는다. IstringStream은 행에서 빈 공백으로 된 경계표식을 뽑아 내는데 쓰인다(이것은 같은 모양을 가지며 임의의 다른 문자처럼 볼것이다.). 행번호는 다음 색인배치표에서 단어와 일치되는것을 추가적으로 기입하는데 첨부된다(단어가 처음으로 출현할 때 식 concordance[word]는 단어와 배치표에 있는 지정벡토르를 포함하는 쌍자료를 삽입한다.). 순환의 끝에서 배치표를 통과하는데 반복자를 리용하며 매개 배치표등록상태를 출력해 낸다.

```
# include <iostream>
# include <fstream>
# include <sstream>
# include <map>
# include <string>
# include <vector>
using namespace std;

   // Output a pair entry: word followed by 1ine numbers
ostream & operator<<( ostream & out, const pair<string. vector<int> > & rhs )
{
      out << rhs.first « ": " « ＼ t' << rhs.second[ 0 ];
      for( int i = 1; i < rhs.second.size( ); i++ )
           out << ", " << rhs.second [ i ];
      return out;
}
int main( int argc, char *argv[ ] )
{
     if( argc != 2 )
     {
           cerr « "Usage: " << argv[ 0 ] << " filename" << endl;
           return 1;
     }
```

```
    ifstream inFi1e( argv[ 1 ] );
    if( ! inFi1e )
    {
        cerr << "Cannot open " << argv[ 1 ] << endl;
        return 1;
    }
    typedef map<string, vector<int> > wordmap;wordmap concordance;
    string oneLine, word;
      // Read the words; add them to wordmap
    for( int lineNum = 1; get1ine( inFile, oneLine ); lineNum++ )
    {
        istringstream st( oneLine );
        while( st >> word )
            concordance[ word ].push_back( lineNum );
    }
        // Output the words
    wordmap:: iterator itr;
    for( itr = concordance.begin( ); itr != concordance.end( ); itr++ )
        cout << *itr << endl;
    return 0;
}
```

프로그람 A-8. STL을 리용한 색인프로그람

프로그람에서 연산자 <<기능은 쌍으로 된 정보단어는 first 자료성원에 기억되여 있고 행번호벡토르는 단어와 second 자료성원에 기억되여 있다. 프로그람은 첫 행번호를 특이한 경우처럼 취급하는데 이것은 한편 프로그람 A-2에서 작성한 프로그람과 류사한데가 있다. 언급할것은 연산자 <<는 행번호의 목록이 비지 않았다는것을 보여 주는데 이것은 코드의 서술없이 담보할수 있다.

2. STL을 쓰지 않는 방안

STL을 리용하지 않고 그와 류사하게 실행시키는 다른 방법은 개선된 클라스를 리용하는것이다. 여기서는 어느 정도의 복잡하고 긴 프로그람을 작성할 때의 세가지 기본차이를 고찰하려고 한다.

① text클라스는 직접 배치표를 실행하지 못한다. 대신 열쇠건과 같이 기호렬로 된 기호렬과 목록을 저축하는 탐색나무를 리용해야 한다.

② 목록클라스는 쉽게 련결되며 끝에 삽입하는 구축방법을 가지고 있지 않다. 그래서 word Entry는 이 련결목록에서 마지막기입을 표시하는 반복자를 가져야 한다.

③ 나무반복자는 없다(련습문제 4-11, 4-13은 비록 작업하도록 요구하지만). printTree방법이 있다. 이 방법은 wordEntry객체가 우에서 본 연산자 <<의 기능으로 제공할것을 요구한다.

수정이 된 프로그람을 두개 부분으로 보여 주었다. 첫 프로그람은 프로그람 A-9에 보여 주었다. 여기서는 #include정의부와 wordEntry클라스를 보여 주고 있다. STL서고를 쓸수 없으며 기호렬의 새로운 변종을 쓸수 없다고 가정하면 istringstream대신 istrstream이 쓰인다. 이것은 머리부파일 strstream.h(일부 체계에서는 strstrea.h라고도 쓰임)를 요구한다. 우에서 설명한것처럼 wordEntry는 세개의 자료성원 즉 단어와 행목록, 행목록에서 마지막위치를 표시하는 반복자를 포함한다. 좀 더 복잡한것은 목록지적자를 리용하며 한편 그의 반복자는 목록의 끝에 행번호를 추가하기때문에 이 자료성원의 하나가 변화되며 하여 word Entry(되돌려 주는)의 불변성을 위반하게 된다. 이로부터 지적자의 값은 변화시키지 않는데 비록 지적된 목록의 상태가 변하더라도 지적자를 리용하는것은 불변안전하다. 또한 연산자 <와 연산자==를 쓰는데 이 기능은 단어구성에 대응하는 기능을 쉽게 찾는다.

main프로그람은 프로그람 A-10에 보여 주며 프로그람 A-8의 main프로그람과 아주 류사하다. 새로운 변종(version)에서 entry는 탐색에 리용되는 wordEntry객체이다. 단어배치표에서 단어를 찾기 위해 string을 탐색하도록 Entry.word를 설정하고 탐색을 진행한다.

```
# inc1ude    <iostream.h>
# include <fstream.h>
#ifdef unix
      # include <strstream.h>
# e1se
      # inc1ude <strstrea.h>
# end if
# inc1ude "mystring.h"
# inc1ude "AvlTree.h"
# include "LinkedList.h" -
class WordEntry
     {
  public:
          WordEntry( ) : word( "" ), 1ines( NULL )
               { }
          bool operator<( const WordEntry & rhs ) const
               { return word < rhs.word; }
          bool operator = = ( const WordEntry & rhs ) const
               { return word = = rhs.word; }
          string              word;
          List<int>        *1ines;
          ListItr<int>    *1istEnd;
```

```
};
   // Output a WordEntry: word followed by line numbers
ostream & operator<<( ostream & out, const WordEntry & rhs )
{
     out << rhs.word << ": ";
     if( rhs.lines != NULL && ! rhs.lines->isEmpty( ) )
     {
          Listltr<int> itr = rhs.1ines->first( );
          out << '\ t' << itr.retrieve( );
          for( itr.advance( ); itr.isPastEnd( ); itr.advance( ) )
                out << ", " << itr.retrieve( );
     }
     return out;
}
```

프로그램 A-9. Text클라스를 쓴 색인프로그람 (1)

탐색결과는 Match이며 일치되는 Word Entry객체를 표현하게 된다. 만일 Match가 ITEM_NOT_FOUND이라면 이것은 새 단어임을 의미한다. 그 경우에 WordEntry객체은 현재행을 삽입하며 단어배치표에 WordEntry객체을 삽입하는 list와 listItr를 동적으로 배치함으로써 구축된다. 한편 일치되는 WordEntry객체에 대하여 행목록의 끝에 현재 행번호를 간단히 추가한 다음 목록에서 마지막위치를 표시하는 반복자를 갱신한다.

```
int main( int argc, char *argv[ ] )
{
   if( argc != 2 )
   {
        cerr << "Usage: " << argv[ 0 ] << " filename" << endl;
        return 1;
   }
   ifstream inFi1e( argv[ 1 ] );
   if( ! inFile )
   {
      cerr << "Cannot open " << argv[ 1 ] << endl;
      return 1;
   }
   const wordEntry ITEM_NOT_FOUND; // "" is the word member
   AvlTree<wordEntry> wordMap( ITEM_NOT_FOUND );
   string oneLine;
   wordEntry entry;
        // Read the words; add them to wordMap
   for( int 1ineNum = 1; get1ine( inFile, oneLine ); 1ineNum++ )
   {
      istrstream st( (char *) oneLine.c_str( ) ); // Deprecated
      while( st >> entry.word )
      {
```

```
                const WordEntry & match = wordMap.find( entry );
                    if( match = = ITEM_NOT_FOUND )
                    {
                        // New word: add to map with line number
                         entry.lines = new List<int>;
                         entry.lines->insert( lineNum, entry, lines->zeroth( ) );
                         entry. listEnd = new Listltr<int>( entry.lines->first( ) );
                         wordMap.insert( entry );
               }
               else
               {
                        // Word already in the map; append the line number
                   match.lines->insert( lineNum, *match. listEnd );
                   match.listEnd->advance( );
               }
           }
       }
       wordMap.printTree( );
       return 0:
   }
```

프로그람 A-10. Text클라스를 쓴 색인프로그람(2)

부록 7. 실례: 최단경로계산

두번째 실례에서는 무게없는**최단경로계산**(*Shortest-Path Algorithm*)알고리듬을 실행하게 된다. 주프로그람은 서술하지 않는다(그와 련결된 프로그람들중의 하나에서 찾아 볼수 있다.). 주프로그람작성에 필요되는 모든 클라스의 방법들 즉 그라프에 경계선을 추가하는 방법, 최단경로계산, 최단경로출력 등을 제공한다. 정점이 외부적으로는 string형으로 표현되지만 내부적으로는 Vertex형(STL실행에서는 배치표를 리용하고 STL이 없으면 하쉬표를 리용)으로 배치된다.

중요하게 설명할것은 코드에 지적자를 리용하는것이다. 이 사정은 일부 기본알고리듬사상의 명백성을 약간 늦추게 한다. 이것이 제9장에서 가상코드를 리용해 온 리유로 된다. 만일 지적자를 써서 C++프로그람을 작성해 보면 코드가 제9장의 가상코드와 류사하게 되는데 이것은 이 장에서 본 가상코드가 알고리듬의 기본사상을 표현한것이라는것을 보여 준다.

프로그람 A-11에서 보여 준 Vertex클라스는 프로그람 9-5에서 준 가상코드와 류사하다. 다만 차이는 dist를 초기화하는 방법과 path마당들을 초기화하는 방법 그리고 구축자를 제공해 준다는것이다.

```
Class Vertex
{
      public:
         string                    name;    // Vertex name
         vector<Vertex *>    adj;      // Adjacent vertices
         int                        dist;       // Cost
         Vertex                 *path;     // Previous vertex on shortest path

         Vertex( const string & nm ) : name( nm )
             { reset( ); }
         void reset( )
             { dist = INFINITY; path = NULL; }
};
```

프로그람 A-11. Vertex 클라스(프로그람 9-5와 같다.)

1. STL에 의한 실현

프로그람 A-12는 그라프클라스이다. 여기서는 세개의 공동방법들을 준다. addedge는 그라프에 새로운 경계선을 그린다. unweighted와 printpath는 제9장에서 본 가상코드루틴들과 같다. 앞에서 언급된것처럼 배치표는 string을 vertex로 변환한다. String과 일치하는 Vertex(그에 대한 지적자)를 주는 하나의 방법을 본다. printpath와 ClearAll은 이미 제9장에서 보아 온 프로그람들과 류사하다. 정점(Vertices)들의 목록(지시기들)을 얻는것이 필요하며 그래서 최단경로(Shorest-path)계산을 진행하는 시점에서 그 모든 거리들을 초기화해야 한다. 이것은 Allvertices자료성원이다.

```
typedef map<string,Vertex *> vmap;

/**
  * Graph class interface (STL version).
  */
class Graph
{
   public
        Graph( ) { }
        ~Graph( );
        void addEdge( const string & sourceName, const string & destName );
        void printPath( const string & destName ) const;
        void unweighted( const string & startName );
   private:
        Vertex * getVertex( const string & vertexName );
        void printPath( const Vertex & dest ) const;
        void clearAll( );
```

```
        vmap vertexMap;
        vector<Vertex *> allVertices;
    };
```

프로그람 A-12. 그라프클라스대면부(STL 실행)

```
/**
 * If vertexName is not present, add it to vertexMap.
 * In either case, return a pointer to the Vertex.
 *./
Vertex * Graph::getVertex( const string & vertexName )
{
    vmap::iterator itr = vertexMap.find( vertexName );

    if( itr = = vertexMap.end( ) )
    {
        Vertex *newv = new Vertex( vertexName );
        all Vertices.push_back( newv );
        vertexMap[ vertexName ] = newv;
        return newv;
    }
    return itr->second;
}
```

프로그람 A-13. Vertex에 대한 지시기를 얻는데 Vertex map를 참고(STL실행)

여기서 이 방법들의 실행을 볼수 있다. 새로운 방법 즉 string들로 취급되는 방법들을 실행하는것으로 프로그람이 시작된다. getVertex는 프로그람 A-13에서 보여 준다. Vertex를 기입하기 위하여 배치표를 참고한다. 만일 vertex가 없다면 새로운 Vertex를 창조하고 배치표를 갱신해야 한다. addEdge는 프로그람 A-14에서 보여 준것처럼 거의나 적게 취급된다. 일치하는 Vertex를 기입해 넣고 다음 린접목록을 갱신한다. 프로그람 A-14에서 보면 printpath는 공동방법이다. 목적하는 Vertex를 얻어서 그것이 존재하는가 그리고 구할수 있는가를 확인한 다음 작성된 printpath루틴을 재귀적으로 호출한다.

```
/**
 *   Add an edge to the graph.
 */
void Graph::addEdge( const string & sourceName,
                     const string & destName )
{
    Vertex * v = getVertex( sourceName );
    Vertex * w = getVertex( destName );
```

```
        v->adj.push_back( w );
}

/**
 * Public routine to print the path to destName
 * after running the shortest-path computation.
 */
void Graph: :printPath( const string & destName ) const
{
    vmap::const_iterator itr = vertexMap.find( destName );

    if( itr = = vertexMap.end( ) )
    {
       cout << "Destination vertex not found" << end1;
       return;
    }
    const Vertex & w = *itr->second;
    if( w.dist = = INFINITY )
        cout << destName << " is unreachable";
    else
        printPath( w );
    cout << endl;
 }
```

프로그람 A-14. 새로운 경계선을 추가하며 경로를 출력하는 공동방법: printpath는 우에서 적재된 재귀루틴을 호출

최단경로계산을 진행하기에 앞서 거리를 무한대로 재설정해야 하며 path전체(이것은 이 공정을 수행하는데 충분하지 못하여 매개의 새로운 계산을 하기전에 설정을 수행해야 한다.)를 지워 버려야 한다. 이것은 프로그람 9-6에 준 가상코드와 같다. 이것을 실현하자면 매개의 Vertex에 대하여 reset산법을 호출해야 한다. 프로그람 A-15는 이 프로그람이 수행되는 과정을 보여 주었는데 여기서는 보통의 수법으로 정점벡토르에 대한 반복을 진행해야 한다. 프로그람 A-15는 또한 graph해체자(destructor)를 보여 준다.

```
/**
 *   Initialize alt Vertex objects prior to running unweighted.
 */
void Graph: :c1earA1l( )
{
   for( int i = 0; i < a11Vertices.size( ); i++ )
       a11Vertices[ i ]->reset( );
}
```

```
/**
 * Destructor: reclaim all dynamically allocated Vertex objects.
 */
Graph::~Graph( )
{
        for( int i = 0; i < allVertices.size( ); i++ )
                delete allvertices[ i ];
}
```

프로그람 A-15. 거리의 초기화와 해체자

Printpath방법을 프로그람 A-16에서 보여 주었다. 이것은 프로그람 9-7에서 준 가상코드와 같다.

```
/**
 * Recursively print path to dest.
 */
void Graph: :printPath( const Vertex & dest ) const
{
    if( dest.path != NULL )
    {
         printPath( *dest.path );
         cout << " to ";
    }
    cout << dest.name;
}
```

프로그람 A-16. 실제최단경로를 출력하는데 리용되는 재귀호출루틴

마지막으로 프로그람 A-17에서 unweighted방법을 보여 준다. 이 코드는 프로그람 9-4에 준 가상코드에 기초한다. 언급할것은 목록은 대기렬을 실행시키는데 front, pop_front, push_back를 리용한다. 코드자체에는 새로운 내용이 적다는것을 설명한다. 반복은 이미 중복하여 써온것과 같은 기술을 리용한다.

```
/**
* Run the unweighted single-source shortest-path algorithm with startName
* as the source vertex. Line numbers correspond to program 9-4.
*/
void Graph::unweighted( const string & startName )
{
    vmap: :iterator itr = vertexMap.find( startName );
```

```
            if( itr = = vertexMap.end( ) )
            {
                cerr « startName << " is not a vertex in this graph" << endl;
                Return;
            }
            c1earA11( );
            Vertex *start = itr->second;
            1ist<Vertex *> q;
/*1*/       q.push_back( start );
/*2*/       start->dist = 0;
/*3*/        whi1e( !q.empty( ) )
             {
/*4*/            Vertex *v = q.front( ); q.pop_front( );
/*6*/            for( int i = 0; i < v->adj.size( ); i++ )
                 {
                       Vertex *w = v->adj[ i ];
/*7*/                  if( w->dist = = INFINITY )
                       {
/*8*/                        w->dist = v->dist + 1;
/*9*/                        w->path = v;
/*10*/                       q.push_back( w );
                       }
                 }
             }
          }
```

프로그람 A-17. 무게붙은최단경로계산

2. STL을 쓰지 않는 방안

본문클라스를 쓴 실행은 STL이 기능적으로는 더 구축되여 있기때문에 STL을 쓰지 않은것보다 더 많은 작업이 요구된다. 묶음에는 배치표가 없기때문에 배치표대신 하쉬표 클라스를 더 리용해야 한다. 이것은 하쉬표가 정점이름(string과 같은)을 포함하는 MapEntry객체와 그것이 배치된 Vertex를 보존해야 한다는것을 의미한다. 하쉬표는 열쇠로서 vertex이름을 리용하게 된다.

프로그람 A-18은 mapEmpty클라스를 보여 준다. 이 클라스는 구축자와 하쉬기능(대역구역에서)과 연산자 ==의 수행, 연산자 :=의 수행을 포함하고 있다. 모든 3가지 기능들은 VertexName성원을 열쇠로 리용한다.

```
/**
 * Entry in the map: stores a string,Vertex* pair.
 */
```

```
class MapEntry
{
  public:
    string vertexName;
    Vertex *storedVertex;
    MapEntry( const string & name = "", Vertex * v = NULL )
      : vertexName( name ), storedVertex( v ) { }
    bool operator!=( const MapEntry & rhs ) const
      { return vertexName != rhs.vertexName; }
    bool operator= =( const MapEntry & rhs ) const
      { return vertexName = = rhs.vertexName; }
};

int hash( const MapEntry & x, int tableSize )
{
      return hash( x.vertexName, tableSize );
}
```

프로그램 A-18. 의 STL의 Graph를 리용하지 않고
하쉬표에서 배치표쌍을 기억시키는데 쓰이는
Map Entry클라스

Graph클라스를 위한 대면부는 프로그람 A-19에 보여 주었다. 이것과 STL변종과의 차이는 배치표대신 하쉬표를 리용한다는것이며 AllVertices는 벡토르대신 목록으로 표시되며 numVertices에서(우리의 목록이 이 기능을 제공하지 못하므로) 정점의 수를 보존한다는것이다. 여섯가지 방법들중에서 printpath는 다시 작성해야 한다.

```
/**
 *  Graph class interface (non-STL version).
 */
class Graph
{
  public:
    Graph( ) : vertexMap( MapEntry( ) ), numVertices( 0 ) { }
    ~Graph( );
    void addEdge( const string & sourceName, const string & destName );
    void printPath( const string & destName ) const;
    void unweighted( const string & startName );
  private:
    Vertex *getVertex( const string & vertexName );
    void printPath( const Vertex & dest ) const;
    void clearAll( );
    HashTab1e<MapEntry> vertexMap;
    List<Vertex *> allVertices;
    int numVertices;
```

```
        const MapEntry ITEM_NOT_FOUND;
};
```

프로그람 A-19. 그라프클라스에서 대면부

GetVertex는 프로그람 A-20에서 보여 주는데 부록 B의 2에서 본것과 같은 기술을 리용한다. 이 열쇠에 맞는 entry객체를 보존하고 MapEntry match를 포함하도록 하기 위해 하쉬표를 참고한다. 그라프응용프로그람인 경우에 만일 ITEM_NOT_FOUND와 일치되면 새 정점을 가지게 되며 이것은 하쉬표에 추가되여야 한다.

```
/**
 *   If vertexName is not present, add it to vertexMap.
 *   In either case, return a pointer to the Vertex.
 */
Vertex * Graph::getVertex( const string & vertexName )
{
        static MapEntry entry;
        entry.vertexName = vertexName;

        const MapEntry & match = vertexMap.find( entry );
        if( match = = ITEM_NOT_FOUND )
        {
                entry.storedVertex = new VertexC vertexName );
                allVertices.insert( entry.storedVertex, a11Vertices.zeroth( ) );
                numVertices++;
                vertexMap.insert( entry );
                return entry.storedVertex;
        }
        return match.storedVertex;
}
```

프로그람 A-20. 정점에 대한 지적자값을 얻는데 VertexMap 참고

프로그람 A-21은 addEdge와 공동 printpath의 기능을 보여 주는데 이것들은 사실 STL을 써서 작업한 프로그람 A-14의 실행과 류사하다. 여기서 addEdge는 V대상의 앞에 W를 삽입하며 끝이 아니라 린접목록에 삽입한다.

```
/**
 *   Add an edge to the graph.
 */
void Graph::addEdge( const string & sourceName,
                     const string & destName )
```

```
{
        Vertex * v = getVertex( sourceName );
        Vertex * w = getVertex( destName );
        v->adj.insert( w, v->adj,zeroth( ) );

  }
  /**
   *  Public routine to print the path to destName
   *  after running the shortest-path computation.
   */
  void Graph::printPath( const string & destName ) const
  {
        const MapEntry,,& match = vertexMap.find( MapEntry( destName ) );
        if( match = = ITEM_NOT_FOUND )
        {
            cout << "Destination vertex not found" << endl;
            return;
        }
        const Vertex & w = *match. storedVertex;
        if( w.dist = = INFINITY )
            cout << destName << " is unreachable";
        else
            printPath( w );
        cout << endl;
  }
```

프로그람 A-21. 새 경계를 추가하며 통로를 출력하는 공동방법

프로그람 A-22에서 clearAll은 매개의 Vertex에 대하여 reset방법을 반복실현하는 기능이다. 이것이 최단경로계산인데 프로그람 A-23에서는 제9장에서 본 가상코드와 프로그람 A-17에서의 STL실행에 대하여 요약해서 보여 주었다.

```
/**
 * Initialize all Vertex objects prior to running unweighted.
 */
void Graph::c1earAl1( )
{
        ListItr<Vertex *> itr;
        for( itr = a11Vertices.first( ); !itr.isPastEnd( );
                itr.advance( ) ) itr.retrieve( )->reset( );
}
/**
 * Destructor: reclaim a11 dynamicany allocated Vertex objects.
 */
Graph::c1ear( )
{
        ListItr<Vertex *> itr;
```

```
        for( itr = a11Vertices.first( ); !itr.isPastEnd( ); itr.advance( ) )
            delete itr.retrieve( );
}
```

프로그램 A-22. 거리의 초기화, 해체자를 표시
(프로그램 9-6 과 같다.)

```
   /**
     * Run the unweighted single-source shortest-path algorithm with startName
     * as the source vertex. Line numbers correspond to program 9-4.
     */
    Void Graph: :unweighted( const string & startName )
    {
        c1earA11( );
        const MapEntry & match = vertexMap.find( MapEntry( startName ) );
        if( match = = ITEM_NOT_FOUND )
        {
            cout << startName << " is not a vertex in this graph" << end1;
            return;
        }
       Vertex *start = match. storedVertex;
       Queue<Vertex *> q( numVertices );
/*1*/  q.enqueue( start );
/*2*/  start->dist = 0;
/*3*/  whi1e( !q.isEmpty( ) )
       {
/*4*/          Vertex *v = q.dequeue( );
               ListItr<Vertex *> itr;
/*6*/          for( itr = v->adj,first( ); !itr.isPastEnd( ); itr.advance( ) )
               {
                  Vertex *w = itr.retrieve( );
/*7*/             if( w->dist == INFINITY )
                  {
/*8*/                     w->dist = v->dist + 1
/*9*/                     w->path = v;
/*10*/                   q.enqueue( w );
                  }
               }
       }
    }
```

프로그램 A-23. 무게없는최단경로계산(프로그램 9-4 와 같다.)

부록 8. STL의 기타 특성

STL은 많은 응용프로그람에서 아주 유용하게 리용할수 있는 강력한 서고이다. 여기서는 STL의 기본골조만을 서술하였다. STL은 이외에도 여러가지 흥미 있는 구조들을 포함하고 있는데 그것들은 다음과 같다.

- 우선권대기렬을 위한 조작
- 열쇠들을 중복렬거할수 있는 다중모임과 다중배치표
- 여러가지 알고리듬(실례로 복사, 교체, 변환, 섞어놓기, 선택, 정렬, 병합)
- 술어에 기초한 탐색알고리듬–임의의 속성에 맞는 대상용기들을 탐색
- 교체반복자
- 강력한 문자입출력렬의 반복자

부록 B. 벡토르와 문자렬클라스

이 부록에서는 STL이 제공하지 않는 콤파일러를 위한 **벡토르**(*vector*)**와 문자렬(***string*) 의 일부 실행에 대하여 설명한다.

부록 1. 1차클라스와 2차클라스객체의 비교

프로그람작성언어를 학습하는 콤퓨터부분 과학자들은 흔히 언어구조를 **1차클라스객체**(*first-class objects*)혹은 **2차클라스객체**(*second-class objects*)로 설계를 시작한다. 이 술어들에 대해서는 정확히 말하기는 좀 어렵지만 기본사상은 1차클라스객체는 특이한 경우가 없이 모두를 《보통의 방법》으로 처리한다는것이며 이와는 반대로 2차클라스객체는 제한된 방법으로만 처리할수 있는것이다.

그러면 《보통의 방법》이란 무엇인가. C++에서 특이한 경우 여기서는 값주기연산자(=), 복사를 실현하기 위한 복사구축자를 포함하며 기억기조작을 실현하는 해체자와 비교연산자==와 같은 비교처리를 진행하는 의미가 있는것들을 포함한다. 1차클라스대상들은 특이한 경우에 대하여 크게 걱정하지 않고도 쓸수 있는 일반형판인수들로 리용될수 있다.

C**문자렬**은 값주기조작과 비교조작이 사실 그 실행에서 표준적으로 기대하는것처럼 진행되지 못하므로 특이한 경우로 조종되여야 한다. C형배렬에 대해서도 마찬가지인데 값주기조작은 완전한 배렬복사를 해내지 못한다.

본문에서는 배렬이나 기호렬대신에 1차클라스로 론법을 제공하는 벡토르와 기호렬 클라스를 리용한다. 이 클라스들은 STL의 부분이며 또한 C++의 부분이다. 어쨌든 많은 콤파일러들은 아직 이러한 클라스들을 지원하지 못한다. 여기서는 자체변종을 제기하며 조작에서 2차클라스의 호환품이 어떻게 만들어 지는가를 설명한다. 이 클라스들은 클라스로 장비된 2차클라스의 특성을 리용하여 실행한다. 이 항목들은 1차객체를 리용하는 사용자들이 정통하지 못하고 또 숨겨 져 있기때문에 2차클라스형으로 접수할수 있는 용법이다.

부록 2. 벡토르클라스

벡토르클라스는 배렬의 첨수화, 재크기화, 복사기능을 제공하며 첨수의 범위를 검사(STL변종에는 그런 기능이 없다.)한다. 이 변종은 우에서 본 색인범위검사를 제외하고는 STL변종만큼 효과적이다. 이 클라스는 기호 NO_CHECK를 리용하며 이것이 정의되였다면 범위검사코드가 번역되지 않게 한다. 모든 콤파일러는 콤파일지령의 일부분으로 기호들을 정의하는데 따라 선택적으로 번역을 진행한다. 즉 지령항목에 따라 콤파일문서를 검사한다. 본문에서 모든 프로그람은 이 벡토르를 리용하며 STL변종이 대신 리용되더라도 이 벡토르클라스에서 모든 성원함수들은 STL변종으로 표시된다.

벡토르클라스는 기준언어의 배렬(객체들)을 자료성원과 같이 기억시키는 방법으로 실행된다. 기준언어에서 배렬은 《2차클라스》객체이며 배렬객체를 기억시키는데 필요한 기억기가 충분히 되도록 하는데 지시기를 리용한다. 기준언어배렬은 지시기로 표현되며 배렬의 크기는 알수 없고 서로 다른 변수로 보존된다. 배렬을 위한 기억구역은 new[ ]연산자를 호출함으로써 배당된다. 이것은 구축자에서와 값주기명령문 그리고 재크기화조작에서 진행된다. 기억기는 Delete[]에 의해 해방된다. 이것은 구축자와 값주기명령문에서 제기된다.(그것은 값주기에서 이전 배렬은 새 배렬이 배치되기전에 해방되며 재크기화를 실현하는동안 낡은 배렬은 새배렬이 배치된후 해방되기때문이다.).

클라스대면부는 프로그람 B-1에서 보여 주며 함수에 대한 무질서한 호출을 피하도록 하기 위하여 하나의 려객선기능을 수행한다. 콤파일러는 이 기능과 직접 련결할수 있다. 사실 이것은 크게 리용가치가 적으나 빠른 벡토르조작은 임의의 응용프로그람에서 극히 중요한것이다. 나머지성원함수들은 프로그람 B-2에 보여 주었다.

```
#ifndef     _VECTOR_H_
#define     _VECTOR_H_
class ArrayIndexOutOfBounds { }; // An exception class
/**
* vector class interface. Supports construction with an initial
```

```
* size (default is 0), automatic destruction, access of the current size,
* array indexing via [], deep copy, and resizing.
* Object must have zero-parameter constructor and operator=.
* Index range checking is performed unless NO_CHECK is defined.
*/
template <c1ass Object>
class vector
{
    public:
        explicit vector( int theSize = 0 ) : currentSize( theSize )
            { objects = new Object[ currentSize ]; }
        vector( const vector & rhs ) : objects( NULL )
            { operator=( rhs ); }

        ~vector( )
         { delete [ ] objects; }
        Int size( ) const
          { return currentSize; }
        Object & operator[] ( int index )
         {
#ifndefNO_CHECK
             if( index < 0        index >= currentSize )
                 throw ArrayIndexOutOfBounds( );
#endif
            return objects[ index ] ;
        }

      const Object & operator[]( int index ) const
      {
#ifndef NO_CHECK
            if( index < 0 || index >= currentSize )
                throw ArrayIndexOutOfBounds( );
#endif
            return objects[ index ];
      }

      const vector & operator=( const vector & rhs );
      void resize( int newSize );
    private:
      int currentSize;
      Object *objects;
  };
```

프로그램 B-1. vector.h

```
#include "vector.h"

template <c1ass Object>
const vector<0bject> &
vector<0bject>::operator=( const vector<0bject> & rhs)
{
      if( this != &rhs )                                    // Alias test
      {
            delete [ ] objects;                             // Reclaim old array
            currentSize = rhs.size( );                      // Copy size member
            objects = new Object[ currentSize ];            // Allocate new array
            for( int k = 0; k < currentSize; k++ )          // Copy the elements
                  objects[ k ] = rhs.objects[ k ];
      }
     return    *this;                                       // Return reference to self
}

template <c1ass Object>
void vector<0bject>::resize( int newSize )
{
             object *oldArray = objects;                    //save loc. Of old array
             int numToCopy = newSize<currentSize ?          //computer number
                                NewSize : currentSize;      // of copied items

              objects = new Object[ newSize ];              // Allocate new array
           currentSize = newSize;                           // Set new size
               for( int k = 0; k< numToCopy; k++ )          // Copy the elements
                      objects[ k ] = oldArray[ k ];
              delete [ ] oldArray;                          // Reclaim old array
}
```

프로그람 B-2. vector.cpp

부록 3. 기호렬클라스

C++는 두개의 기호렬형 즉 **C형의 기호렬**(*C-style string*)(C프로그람언어로부터 나온)과 STL 그리고 일부 언어에 추가된 **기호렬클라스**(*string class*)를 가진다. 만일 콤파일러가 기호렬클라스를 가지고 있다면 이것을 리용하게 하는데 이것은 매우 효률적이다. 한편 C형의 기호렬을 리용하든지 아니면 자체가 제공하는 기호렬을 리용하는데는 선택이 필요하다.

C형의 기호렬은 기호렬을 표시하는데 기호들의 배렬을 리용한다. 마지막기호는 령끝기호로서 특이한 기호 ＼0를 가진다. 이로부터 기호렬 《abc》는 char의 배렬로 기억되는데 첫 네개의 배렬위치에 기호 a, b, c, ＼0을 포함하게 된다. 마지막에 뒤따르는 령끝기

호 ＼0는 기호렬로 보지 않는다. 이로부터 배렬의 이름은 곧 지적자이며 C형의 기호렬들은 1차클라스객체와 같이 취급될수 없다. 그러면서도 기호렬을 복사하는데는 strcpy라고 하는 기능을 리용해야 한다. 이때 사용자는 거기에 복사되고 있는 기호렬을 기억시키는데 기억공간이 충분히 크다는 담보를 해야 한다. 이것으로 처리되는 C형기호렬들은 오유가 나오기 쉽다. C형기호렬을 비교하는데 ctrcmp기능을 리용한다. 기호렬에서는 배렬을 색인함으로써 개별적인 기호들을 호출할수 있으나 색인은 검사되지 않는다.

또한 C형기호렬은 조작이 오래 걸릴뿐아니라 그것들은 형판을 가지고 작업하지 않는다. 실례로 정렬알고리듬형판은 연산자 <가 요소들을 순서대로 배렬하도록 한다. 그러나 연산자 <는 C형기호렬에 대하여 기호렬이 기억되여 있는 기억구역을 간단히 비교하며 이로부터 배렬의 이름은 지적자변수이다. 결과적으로 C형기호렬은 가능한껏 쓰는것을 피해야 하며 string클라스는 1차클라스특성과 비슷하게 리용된다.

string클라스대면부를 프로그람 B-3에 보여 주었다. 머리부파일 string.h와의 충돌을 피하기 위해 mystring.h에 련결부를 기억시킨다. 다음 세개의 자료성원들은 C형의 기호렬과 기호렬의 길이, 배렬의 크기를 기억시킨다. 배렬의 크기는 적어도 길이보다는 하나만큼 더 크다. 여기서는 두개의 호출자(C_str와 length)를 제공하는데 이것들은 C형기호렬과 기호렬의 길이이다. 연산자 +=는 rhs를 현재 기호렬에 추가한다. 비클라스기능들의 모임은 또한 I/O와 비교를 위하여 제공되며 비교된다. I/O기능들은 string이 첫 파라메터가 아니므로 클라스성원이 아니다.

비교기능은 클라스성원처럼 정확히 실행되지 않는다. 클라스밖에서 이 기능들을 실행할 때에는 비교연산자의 왼쪽켠에 C형의 기호렬 혹은 string을 써야 한다. 비교연산자에서 하나의 연산수가 C형의 기호렬이라면 림시string은 구축(string구축자를 호출함으로써)하여야 한다. 그러므로 str1과 str2가 기호렬들이면 다음의 식들은 옳다. 즉 str1== str2, str1==“ab”, “ab”== str2 만일 비교기능이 클라스성원들(이 경우 비교기능의 정의는 rhs요소로만 씌여짐)이라면 “ab”== str2로 합법적으로 표시할수 없을것이다.

```
#ifndef _MY_STRING_H_
#define _MY_STRING_H_
#include <iostream.h>
class StringIndexOutOfBounds { };
class string
{
  public:
        string( const char *cstring = " " );                    // Constructor
        string( const string & str );                           // Copy constructor
        ~string( )                                              // destructor
           { delete [ ] buffer; }
```

```
        const string & operator= ( const string & rhs ); // Copy
        const string & operator+=( const string & rhs ); // Append

        const char * c_str( ) const                         // Return C-style string
           {return buffer;}
        int length( ) const                                  // Return string length
             {return strength}

        char     operator[]( int k ) const;                  // Accessor operator[ ]
        char & operator[]( int k );                          // Mutator operator[ ]

        enum {MAX_LENGTH = 1024};                            // Maximum length for input string
   private:
         char *buffer;                                       // strage for characters
        int strLength;                                       // length of string (# of characters)
        int bufferLength                                     // capacity of buffer
}

ostream & operator<<(ostream & out const string & str);   // Output
istream & operator>>(istream & in string & str);          // Input
istream & getline( istream & in, string & str);                       //read line
bool operator ==( const string & lhs, const string & rhs);            //compare==
bool operator !=( const string & lhs, const string & rhs);            //compare !=
bool operator < ( const string & lhs, const string & rhs);            // compare <
bool operator <=( const string & lhs, const string & rhs);            // compare <=
bool operator > ( const string & lhs, const string & rhs);            // compare >
bool operator >=( const string & lhs, const string & rhs);            // compare >=
#endif
```

프로그람 B-3. mystring.h

구축자들은 프로그람 B-4에 보여 주는데 상대적으로 간단하다. 즉 이것들은 세개의 자료성원들을 초기화한다. 값주기연산자(프로그람 B-5)는 보다 더 복잡하고 까다로운데 왜냐하면 그것들은 두가지 문제를 포함하고 있다.

```
#include <string.h>
#include "mystring.h"

string: :string( const char * cstring )
{
     if( cstring == NULL )                                // If NULL pointer
          Cstring = "";                                   // use empty string
     strLength = strlen( cstring );                       // Get length of other string
bufferLength = strLength + 1;                        // Set length with null terminator
     buffer = new char[ bufferLength ];                   // Allocate C-style string
     strcpy( buffer, cstring );                           // Do the copy
```

```
}
string::string( const string & str )
{
    strLength = str.length( );                    // Get length of other string
    bufferLength = strLength + 1;                 // Set length with null terminator
    buffer = new char[ bufferLength ];            // Allocate C-style string
    strcpy( buffer, str.buffer );                 // Do the copy
}
```

프로그램 B-4. string.cpp(I 부류): 구축자

```
const string & string: :operator=( const string & rhs )
{
        if( this != &rhs )                                // Alias test
        {
            if( bufferLength < rhs.length( ) + 1 )        // If not enough room
            {
                delete [ ] buffer;                        // Reclaim old array
                bufferLength = rhs.1ength( ) + 1;         // Compute new size
                buffer = new char[ bufferLength ];        // Allocate new array
            }
            strLength = rhs.Length( );                    // Set new length
            strcpy( buffer, rhs.buffer );                 // Do the copy
        }
         return *this;                                    // Return reference to self
}
const string & string::operator+=( const string & rhs)
{
        if( this == &rhs )                                // Alias test: if s+=s
        {

            string copy( rhs );                           // Make a copy of rhs
            return *this += copy;                         // Append copy; avoid alias
        }
        int newLength = length( ) + rhs.length( );        // Compute new length
       if( newLength >= bufferLength )                    // If not enough room
        {                                                 // Begin the expansion:
            bufferLength = 2 * ( newLength + 1 );         // Allocate more room; use
                              // 2x space so repeated calls to += are efficient
            char *oldBuffer=buffer;                       //Save location of old array
            buffer=new char[bufferLength];                //Allocate new array
            strcpy(buffer, oldBuffer);                    //Do the copy
            delete [ ] oldBuffer;                         //Beclaim old array
```

```
    }
    strcpy(buffer+length(), rhs, buffer);                 //Append rhs
    strLength=newLength;                                  //Set new length
    return *this;                                         //Return reference to self
}
```

프로그람 B-5. string.cpp(Ⅱ부류): 값주기연산자

하나는 결과로 얻어 진 기호렬이 적합치 않으면 완충기억기는 확장되여야 한다.두번째는 가명에 대한 조종이 쉽게 되여야 한다. 연산자 +=에서 가명검사가 빠지면 완충기억기의 재크기화가 요구될 때 str+= str에 대하여 효력이 없는 지적자를 창조(즉 이미 지워 진 기억기를 지시하는 지적자의 창조)할수 있다.

연산자 +=의 연산량은 *O(N)*이다. 재크기화를 요구하면 이것은 증가한다. 이런 기억공간의 피해를 없애기 위하여 실지 필요되는 기억공간의 2배만큼 더 큰 새로운 완충기를 준비한다. 이 리치는 재하쉬표만들기(제5장 제5절)에서와 배묶음만들기(련습문제 3-29, 3-30)에서 리용된것과 같다.

배렬색인만들기조작자는 프로그람 B-6에 보여 주었다. 이것은 예비처리지령을 생략한것을 제외하고 벡토르클라스에서 본 조작자와 같다. 프로그람 B-7은 I/O조작자를 보여준다. 먼저 입력을 위하여 MAX_LENGTH의 한계를 가정한다. 이것은 C형기호렬을 리용하는데서 일반적인 처리방도를 례증하고 있다.

```
char & string: :operator[ ]( int k )
{
    if( k < 0 || k >= strLength )
        throw StringIndexOutOfBounds( );
    return buffer[ k ];
}
char string::operator[ ]( int k ) const
{
    if( k < 0 || k >= strLength )
        throw StringIndexOutOfBounds( );
    return buffer[ k ];
}
```

프로그람 B-6. string.cpp(Ⅲ부류): 색인만들기연산자

```
ostream & operator«( ostream & out, const string & str )
{return out « str.c_str( );
}
istream & operator»( istream & in, string & str )
{
char buf[ string::MAX_LENGTH + 1 ];
in » but;
str = but;
return in;
}

istream & get1ine( istream & in, string & str )
{
   char buf[ string::MAX_LENGTH + 1 ];
   in.get1ine( but, string::MAX_LENGTH );
      str = but;
      return in;
}
```

프로그램 B-7. string. CPP(IV 부류):
I/O 기능들

이 기능들은 클라스성원들이 아니기때문에 임의의 개별적자료들을 호출할수 없고 호출하지도 못한다는데 대해 주의를 돌려야 한다. 비교연산자는 프로그람 B-8에 보여 주었다. 이것들은 C형태의 기호렬에 대하여 strcmp를 간단히 호출한다. buffer는 개별적인 자료성원이고 그 조작자는 클라스의 성원이 아니기때문에 C기호렬을 얻자면 호출자를 다시 리용해야 한다. 이 기호렬클라스는 효과가 적은데 암시적인 형변환을 진행하는것으로 하여 어느 정도의 신용을 얻게 된다. 이로부터 만일 C형태의 기호렬(혹은 기호렬포함)이 비교연산자나 값주기연산자를 통과하면 림시기호렬이 생긴다는것을 알수 있다. 이것은 실행시에 머리부에 추가할수 있다. 이 문제의 풀이는 C형의 기호렬을 하나의 인수로 하는 추가기능을 작성하는것이다.

```
bool operator==( const string & lns, const string & rhs )
{
    return strcmp( lhs.c_str( ), rhs.c_str( ) ) = = 0;
}
bool operator!=( const string & lns, const string & rhs )
{
    return strcmp( lhs.c_str( ), rhs.c_str( ) ) != 0;
}
```

```
bool operator<( const string & rns, const string & rhs )
{
    return strcmp( lhs.c_str( ), rhs.c_str( ) ) < 0;
}
bool operator<=( const string & lhs, const string & rhs )
{
    return strcmp( lhs.c_str( ), rhs.c_str( ) ) <= 0;
}
bool operator>( const string & lhs, const string & rhs )
{       return strcmp( lhs.c_str( ), rhs.c_str( ) ) > 0;
}
bool operator>=( const string & lhs, const string & rhs )
{
    return strcmp( lhs.c_str( ), rhs.c_str( ) ) >= 0;
}
```

프로그람 B-8. string. CPP(V 부류): 비교연산자들

그래서 대역적인 기능들을 아래의 형태로 추가한다.

```
Bool operator<=(const char * lhs,const string & rhs);
Bool operator<=(const string & lhs,const char * rhs);
```

그리고 클라스성원기능들도 추가한다.

```
const strin & operator<=(const char * rhs);
const strin & operator<=(const char * rhs);
```

이것은 단순char형을 인수로 하는 축적연산자 +=를 추가할수 있게 한다.

색　　인

ㄱ

ㄴ

ㄷ

ㄹ

ㅁ

ㅂ

ㅈ

ㅊ

ㅋ

ㄷ

ㅍ

ㅎ

ㄲ

ㅆ

ㅇ